ИАН СОММЕРВИЛЛ

Инженерия программного обеспечения

6-е издание

Addison-Wesley
ВИЛЬЯМС

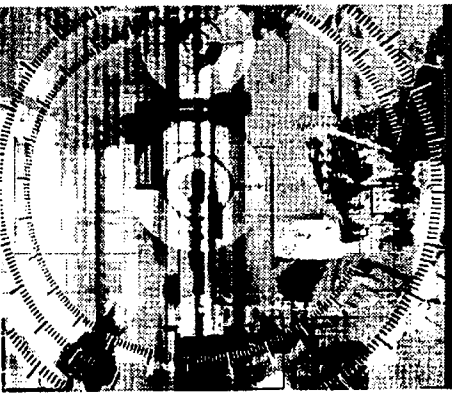
Software Engineering

Ian Sommerville

Addison-Wesley

An imprint of Pearson Education

Harlow, England · London · New York · Reading, Massachusetts · San Francisco · Toronto · Don Mills, Ontario · Sydney · Tokyo · Singapore · Hong Kong · Seoul · Taipei · Cape Town · Madrid · Mexico City · Amsterdam · Munich · Paris · Milan



Инженерия программного обеспечения

6-е издание

Иан Соммервилл

Издательский дом "Вильямс"
Москва • Санкт-Петербург • Киев
2002

ББК 32.973.26-018.2.75
С61
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *А.В. Слепцов*

Перевод с английского канд.физ.-мат.наук *А.А. Минько*,
А.А. Момотюк, Г.И. Сингаевской, В.Д. Яновской

Под редакцией канд.физ.-мат.наук *А.А. Минько*

По общим вопросам обращайтесь в Издательский дом "Вильямс"
по адресу: info@williamspublishing.com, http://www.williamspublishing.com

Соммервилл, Иан.

С61 Инженерия программного обеспечения, 6-е издание. : Пер. с англ. — М. : Издательский дом "Вильямс", 2002. — 624 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0330-0 (рус.)

Данная книга является прекрасным введением в инженерию программного обеспечения. Здесь дана широкая панорама тем инженерии ПО, охватывающих все этапы и технологии разработки программных систем. В семи частях книги представлен весь спектр процессов, ведущих к созданию программного обеспечения: от начальной разработки системных требований и далее через проектирование, непосредственное программирование и аттестацию до модернизации программных систем. Эта книга окажет неоценимую поддержку студентам и аспирантам, изучающим дисциплину "Инженерия программного обеспечения", а также будет полезна тем специалистам по программному обеспечению, которые хотят познакомиться с новыми технологиями разработки ПО, такими, как спецификация требований, архитектура распределенных структур или надежность программных систем.

ББК 32.973.26-018.2.75

ISBN 5-8459-0330-0 (рус.)
ISBN 0-201-39815-X (англ.)

Оглавление

Содержание

Предисловие

Программные системы ныне присутствуют повсеместно: практически любые электронные устройства содержат программное обеспечение (ПО) того или иного вида. Без соответствующего программного обеспечения в современном мире невозможно представить индустриальное производство, школы и университеты, систему здравоохранения, финансовые и правительственные учреждения. Многие используют ПО для самообразования или различного рода развлечений. Создание спецификации требований, разработка, модификация и сопровождение таких систем ПО составляет суть технической дисциплины *инженерия программного обеспечения* (software engineering).

Даже простые системы ПО обладают высокой степенью сложности, поэтому при их разработке приходится использовать весь арсенал технических и инженерных методов. Таким образом, инженерия программного обеспечения — это инженерная дисциплина, где разработчики ПО используют теорию и методы компьютерных наук для успешного решения различных нетривиальных задач (но, конечно, не каждый проект ПО в силу различных причин успешно завершается). Большинство современных программ предлагают пользователям большие сервисные возможности для работы с ними — нельзя не заметить реального прогресса в развитии технологии создания ПО за последние 30 лет.

Инженерия программного обеспечения развивается в основном в соответствии с постановкой новых задач построения больших пользовательских систем ПО для промышленности, правительства и оборонного ведомства. С другой стороны, в настоящее время сфера программного обеспечения чрезвычайно широка: от игр на специализированных игровых консолях, а также программных продуктов для персональных компьютеров и Web-ориентированных программных систем до очень больших масштабируемых распределенных систем. Хотя некоторые технологии, применяемые для построения пользовательских программных систем, универсальны (например, объектно-ориентированные методы), новые технологии создания программного обеспечения развиваются с учетом различных типов ПО. Невозможно охватить в одной книге все технологии создания ПО, поэтому я сконцентрировал основное внимание не на методах разработки отдельных программных продуктов, а на универсальных технологиях и методах проектирования и построения больших масштабируемых программных систем.

Хотя данная книга запланирована как общее введение в инженерию программного обеспечения, в ней отражены мои собственные интересы к критическим системам и процессу разработки требований для программных систем. Я думаю, что эти вопросы имеют практическую важность в преддверии 21-го столетия, так как благодаря их решению современное ПО сможет удовлетворить реальные нужды пользователей следующего века.

В этой книге я старался дать широкую панораму инженерии программного обеспечения без концентрации внимания на каких-либо специальных методах или средствах. Я испытываю стойкую неприязнь к фанатикам любого рода, будь то академический проповедник, рекламирующий формальные методы и подходы, или продавец ПО, пытающийся убедить меня, что именно его программное средство или метод решит все проблемы создания программного обеспечения. Не существует простых решений задач создания ПО; для этого необходим широкий спектр средств, методов и технологий.

Книги неизбежно отражают мнения и предпочтения авторов. Некоторые читатели конечно же не согласятся с моим мнением или выбором материала для книги. Такое несогласие, т.е. здоровая реакция, вызванная множественностью подходов к инженерии программного обеспечения, и является залогом ее дальнейшего развития. Вместе с тем я надеюсь, что все разработчики программного обеспечения, а также студенты, изучающие дисциплину "Инженерия программного обеспечения", найдут в этой книге что-то интересное для себя.

Отличия от пятого издания

Подобно многим системам программного обеспечения, эта книга растет и изменяется начиная с первого издания, опубликованного в 1982 году. Одной из целей при подготовке настоящего издания было сокращение объема книги. Это привело к определенной реорганизации ее содержания и к глубоким раздумьям о том, какой материал можно удалить для того, чтобы включить новый. В результате объем книги уменьшился на 10% по сравнению с пятым изданием.

- Теперь книга содержит семь частей, а не восемь, как было в предыдущем издании. В них представлено введение в инженерию программного обеспечения, темы разработки требований для ПО, собственно создания программного продукта, разработки критических систем, проверки и аттестации ПО, управления ПО, а также эволюции программных систем.
- Добавлены новые главы, описывающие процесс создания и эксплуатации программного обеспечения, архитектуру распределенных систем, проблемы надежности и наследования ПО. Теме формальной спецификации ПО теперь посвящена только одна глава. Материал по CASE-технологиям сокращен и распределен по нескольким главам. Материал по функциональному проектированию ПО теперь содержит новую главу по наследуемым системам (legacy systems). Главы, описывающие верификацию и аттестацию ПО, объединены в отдельную часть.
- Во все главы внесены изменения, а некоторые существенно переработаны. Повторное использование программных компонентов в настоящем издании рассматривается как тема разработки ПО с повторным использованием программных продуктов совместно с материалом о паттернах и темой разработки ПО, основанной на покомпонентном подходе. Главы по разработке требований,выделены в отдельную часть книги.
- Вводная часть книги содержит четыре главы. В них собран вводный материал, который в пятом издании был разбросан по всей книге. Глава 1 полностью переписана и построена в форме вопросов и ответов, касающихся инженерии программного обеспечения.
- Материал по критическим системам реструктуризирован и объединен таким образом, чтобы темы надежности, безопасности и работоспособности ПО теперь не выделялись как отдельные. Добавлен также небольшой материал о защищенности ПО как атрибуте критических систем.
- В данном издании примеры программ написаны на языке Java, а модели объектов — на UML. Примеры программ на языках Ada и C++ удалены из книги, но доступны на моем Web-узле.

Круг читателей книги

Книга предназначена для студентов, изучающих технологии создания ПО, и специалистов по программному обеспечению, работающих в различных областях индустрии разработки программных систем. Ее можно использовать как основу базового курса по инженерии программного обеспечения или в качестве материала для таких курсов, как углубленные технологии программирования, спецификации ПО, разработка и управление программными системами. Программисты-практики не только найдут в книге много полезного в аспекте общих концептуальных вопросов, но и "освежат" знания по таким прак-

тическим темам, как разработка требований, предъявляемых к ПО, разработка архитектуры программных систем, конструирование надежных систем и управление процессом создания ПО. Кроме того, практические примеры, сопровождающие изложение материала, дадут представление о тех типах программных приложений, с которыми специалист по программному обеспечению может столкнуться в практической деятельности.

Я предполагаю, что читатель имеет базовые знания по программированию и современным компьютерным системам, а также знаком с такими основными структурами данных, как стеки, списки и очереди.

Использование книги как учебного пособия

Материал книги может служить основой для трех курсов по технологии создания программного обеспечения.

1. *Базовый вводный курс по инженерии программного обеспечения.* Для этого можно использовать материал первой части, затем выбрать отдельные главы из остальных частей книги. Это даст студентам общее представление об инженерии программного обеспечения и позволит в дальнейшем подробнее изучить темы, которые их интересуют.
2. *Вводные или углубленные курсы по отдельным темам инженерии программного обеспечения.* На базе материала книги можно создать курсы по спецификации ПО, методам разработки ПО, управлению процессом создания ПО, разработке надежных систем и эволюции ПО. Каждая часть книги может служить основой вводного или углубленного курса по этим темам.
3. *Более углубленные курсы по отдельным темам инженерии программного обеспечения.* Главы книги могут составить основу таких курсов, но необходимо дополнить их материалом из других источников. Дополнительные материалы можно найти на моей Web-странице.

Достоинство настоящей книги состоит в том, что ее можно использовать как основу для многих различных взаимосвязанных курсов. Например, в университете Ланкастера мы используем материал этой книги для вводного курса по инженерии программного обеспечения, для курса по спецификации ПО, курса по разработке критических систем и для курса по управлению разработкой ПО, дополняя его при необходимости материалом из других источников. Книга дает студентам последовательное и достаточно полное представление о технологии создания ПО, так что им не приходится покупать несколько книг по этой тематике.

Книга содержит весь материал, рекомендованный Ассоциацией по вычислительной технике и Институтом инженеров по электротехнике и электронике (ACM и IEEE) в документе *Computing Curricula 2001* (Учебный план по компьютерным дисциплинам 2001), по компоненту *Software Engineering* (Инженерия программного обеспечения) базового набора дисциплин, составляющих основу компьютерных наук. Материал книги также соответствует документу *Software Engineering Body of Knowledge* (Совокупность знаний по инженерии программного обеспечения), который должен был быть опубликован ACM/IEEE в 2001 году.

Web-страница

Моя Web-страница (http://www.software-engin.com) содержит ссылки на материал, который используется в этой книге и будет полезен как преподавателям, так и изучающим инженерию программного обеспечения самостоятельно. Для загрузки доступны отмеченные ниже материалы.

- Руководство для преподавателей, включающее советы по использованию данной книги, рекомендации по аудиторной работе, задания для самостоятельного изучения, решения некоторых примеров из книги. Весь материал представлен в формате Adobe PDF.
- Иллюстративный материал для каждой главы в формате Adobe PDF и Microsoft PowerPoint. Преподаватели могут адаптировать и изменять этот материал в соответствии со своими потребностями.
- Исходные коды программ на языке Java для всех основных примеров из книги, включая дополнительный код, необходимый для компиляции.
- Дополнительный материал, используемый в предыдущих изданиях книги. Также доступны программные примеры на языках Ada и C++, которые иллюстрировали пятое издание.

Web-страница содержит ссылки на статьи по инженерии программного обеспечения, ссылки на другие Web-страницы аналогичной направленности, информацию о книгах по данной тематике и рекомендации для дальнейшего чтения.

Я всегда рад получить отклики на мои книги; вы можете писать мне по адресу электронной почты ian@software-engin.com. Но я заранее приношу свои извинения за то, что из-за отсутствия времени не смогу ответить студентам на вопросы их домашних заданий.

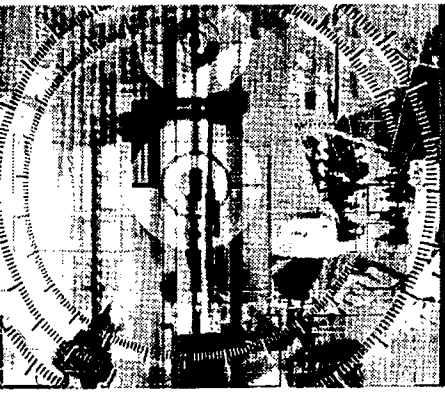
Благодарности

За все годы существования этой книги многие внесли свой вклад в ее эволюцию, и поэтому я хочу поблагодарить каждого, кто высказал замечания по тексту предыдущих изданий и дал свои рекомендации по его улучшению. Я благодарен всем, кто отметил отдельные недостатки в тексте книги и сделал полезные замечания, которые учтены в последнем издании.

Прежде всего за отмеченные недостатки и полезные рекомендации я хочу поблагодарить Энди и Линдсея Гиллесов (Andy Gillies и Lindsey Gillies) из университета Восточной Англии, Джоя Ламберта (Joe Lambert) из университета шт. Пенсильвания, Франка Меддикса (Frank Maddix) из университета Восточной Англии, Нэнси Мед (Nancy Mead) из Института инженерии программного обеспечения (Software Engineering Institute), г. Питтсбург, Крис Прайс (Chris Price) из университета Уэльса, г. Аберистуит, Грегга Ротермела (Gregg Rothermel) из университета шт. Орегон и Гууса Шрейбера (Guus Schreiber) из университета Амстердама. Особую благодарность хочу выразить моим друзьям Рону Моррисону (Ron Morrison) из университета Св. Андрея и Рею Велланду (Ray Welland) из университета Глазго, которые рецензировали предыдущие издания книги и вновь добровольно прочитали текст этого издания.

Особо хотелось бы отметить заслуги моей семьи; они смирились с моим постоянным отсутствием за ужином в те бесконечные дни, когда я заканчивал эту книгу. Благодарю мою жену Энни и дочерей Али и Джейн за прекрасный кофе и терпение.

Иан Соммервилл
Ланкастер, февраль 2000 г.



Инженерия программного обеспечения: обзор

1 Введение

Цели

Цель настоящей главы — дать введение в предмет "инженерия программного обеспечения". Прочитав эту главу, вы должны:

- иметь понятие о том, что такое инженерия программного обеспечения и почему она важна;
- знать ответы на ключевые вопросы, относящиеся к инженерии программного обеспечения;
- понимать этические и профессиональные проблемы, стоящие перед специалистами по программному обеспечению.

Содержание

Современный мир все больше зависит от систем, построенных на основе вычислительной техники. Все больше технических устройств включают в себя элементы вычислительной техники и соответствующего управляющего программного обеспечения (ПО) в той или иной форме. В таких системах стоимость ПО порой составляет большую часть общей стоимости. Более того, стоимостные показатели отраслей, занимающихся производством ПО, становятся определяющими для экономики — как национальной, так и международной.

Целью инженерии программного обеспечения является эффективное создание программных систем. Программное обеспечение абстрактно и нематериально. Оно не имеет физической природы, отвергает физические законы и не подвергается обработке производственными процессами. Такой упрощенный взгляд на ПО показывает, что не существует физических ограничений на потенциальные возможности программных систем. С другой стороны, отсутствие материального наполнения порой делает ПО чрезвычайно сложным и, следовательно, трудным для понимания "объектом".

Инженерия программного обеспечения — сравнительно молодая научная дисциплина. Термин *software engineering*[1] был впервые предложен в 1968 году на конференции, посвященной так называемому кризису программного обеспечения. Этот кризис был вызван появлением мощной (по меркам того времени) вычислительной техники третьего поколения. Новая техника позволяла воплотить в жизнь не реализуемые ранее программные приложения. В результате программное обеспечение достигло размеров и уровня сложности, намного превышающих аналогичные показатели у программных систем, реализованных на вычислительной технике предыдущих поколений.

Оказалось, что неформальный подход, применявшийся ранее к построению программных систем, недостаточен для разработки больших систем. На реализацию крупных программных проектов иногда уходили многие годы. Стоимость таких проектов многократно возрастала по сравнению с первоначальными расчетами, сами программные системы получались ненадежными, сложными в эксплуатации и сопровождении. Разработка программного обеспечения оказалась в кризисе. Стоимость аппаратных средств постепенно снижалась, тогда как стоимость ПО стремительно возрастала. Возникла необходимость в новых технологиях и методах управления комплексными сложными проектами разработки больших программных систем.

Такие методы составили часть инженерии программного обеспечения и в настоящее время широко используются, хотя, конечно же, не являются универсальными. Сохраняется много проблем в процессе разработки сложных программных систем, на решение которых затрачивается много времени и средств. Реализация многих программных проектов сталкивается с подобными проблемами, это дает право некоторым специалистам утверждать, что современная технология создания программного обеспечения находится в состоянии хронического недуга [287].

С другой стороны, возрастает как объем производства программного обеспечения, так и его сложность. Кроме того, сближение вычислительной и коммуникационной техники ставит новые требования перед специалистами по программному обеспечению. Это также является одной из причин возникновения проблем при разработке программных систем, как и то, что многие компании, занимающиеся производством ПО, не уделяют должного внимания эффективному применению современных методов, разработанных в рамках инженерии программного обеспечения. Дело не так плохо, как утверждают скептики, благодаря критике которых высвечиваются болевые точки, способные стать точками роста инженерии программного обеспечения.

---

[1] *Именно этот термин в данной книге мы переводим как инженерия программного обеспечения. Другие существующие переводы (например, программотехника или программная инженерия) нам кажутся не совсем точно отображающими сущность этого понятия.* – Прим. ред.

Я думаю, по сравнению с 1968 годом сделан огромный скачок и развитие инженерии программного обеспечения значительно улучшило современное ПО. Теперь мы лучше понимаем те процессы, которые определяют развитие программных систем. Разработаны эффективные методы спецификации ПО, его разработки и внедрения. Новые средства и технологии позволяют значительно уменьшить усилия и затраты на создание больших и сложных программных систем.

Специалисты по программному обеспечению могут гордиться такими достижениями. Без современного сложного ПО было бы невозможно освоение космического пространства, не существовало бы Internet и современных телекоммуникаций, а все транспортные средства и виды транспортировки были бы более опасными и дорогостоящими. Инженерия программного обеспечения достигла многого за свою пока еще короткую жизнь, и я уверен, что ее значение как зрелой научной дисциплины еще более возрастет в XXI столетии.

1.1. Вопросы и ответы об инженерии программного обеспечения

Этот раздел построен в виде ответов на некоторые основные вопросы, касающиеся инженерии программного обеспечения и отображающие мой собственный взгляд на эту дисциплину. Я использовал формат "списка FAQ" (Frequently Asked Questions — часто задаваемые вопросы). Такой формат обычно применяется в группах новостей Internet, предлагая новичкам ответы на часто задаваемые вопросы. Надеюсь, что подобный подход будет эффективен в качестве краткого введения в предмет инженерии программного обеспечения.

Вопросы и ответы, подробно рассматриваемые в этом разделе, компактно представлены в табл. 1.1.

Таблица 1.1. Часто задаваемые вопросы об инженерии программного обеспечения

Вопрос	Ответ
Что такое программное обеспечение (ПО)?	Это компьютерные программы и соответствующая документация. Программные продукты разрабатываются или по частному заказу, или для продажи на рынке программных продуктов
Что такое инженерия программного обеспечения?	Это инженерная дисциплина, охватывающая все аспекты разработки программного обеспечения
В чем различие между инженерией программного обеспечения и компьютерной наукой?	Компьютерная наука — это теоретическая дисциплина, охватывающая все стороны вычислительных систем, включая аппаратные средства и программное обеспечение; инженерия программного обеспечения — практическая дисциплина создания и сопровождения программных систем
В чем различие между инженерией программного обеспечения и системотехникой?	Системотехника охватывает все аспекты разработки вычислительных систем (включая создание аппаратных средств и ПО) и соответствующие технологические процессы. Технологии инженерии программного обеспечения являются частью этих процессов
Что такое технологический процесс создания ПО?	Это совокупность процессов, ведущих к созданию или развитию программного обеспечения

Окончание табл. 1.1

Вопрос	Ответ
Что такое модель технологического процесса создания ПО?	Формализованное упрощенное представление технологического процесса создания ПО
Какова структура затрат на создание ПО?	Примерно 60% от общих затрат на создание ПО занимают затраты непосредственно на разработку ПО и 40% – на его тестирование и отладку. Для программных продуктов, разрабатываемых по заказу, стоимость тестирования и отладки часто превышает стоимость разработки продукта
Что такое методы инженерии программного обеспечения?	Это структурные решения, предназначенные для разработки ПО и включающие системные модели, формализованные нотацию и правила проектирования, а также способы управления процессом создания ПО
Что такое CASE (Computer-Aided Software Engineering – автоматизированное проектирование и создание ПО)?	Это программные системы, предназначенные для автоматизации процесса создания ПО. CASE-средства часто используются в качестве основы методов инженерии программного обеспечения
Каковы признаки качественного ПО?	Программные продукты должны удовлетворять требованиям функциональности и эффективности (с точки зрения пользователя), а также быть надежными, удобными в эксплуатации и иметь возможности для модернизации
Какие основные проблемы стоят перед специалистами по программному обеспечению?	Проблема наследования ранее созданного ПО, проблема все возрастающей разнородности программных систем и проблема, порожденная требованием уменьшения времени на создание ПО

1.1.1. Что такое программное обеспечение

Многие отождествляют термин *программное обеспечение* с компьютерными программами. Это весьма ограниченное представление. Программное обеспечение – это не только программы, но и вся сопутствующая документация, а также конфигурационные данные, необходимые для корректной работы программ. Программные системы состоят из совокупности программ, файлов конфигурации, необходимых для установки этих программ, и документации, которая описывает структуру системы, а также содержит инструкции для пользователей, объясняющие работу с системой, и часто адрес Web-узла, где пользователь может найти самую последнюю информацию о данном программном продукте.

Специалисты по программному обеспечению разрабатывают программные продукты, т.е. такое ПО, которое можно продать потребителю. Программные продукты делятся на два типа.

1. *Общие программные продукты.* Это автономные программные системы, которые созданы компанией по производству ПО и продаются на открытом рынке программных продуктов любому потребителю, способному их купить. Иногда их называют "коробочным ПО". Примерами этого типа программных продуктов могут служить системы управления базами данных, текстовые процессоры, графические пакеты и средства управления проектами.

2. *Программные продукты, созданные на заказ.* Это программные системы, которые создаются по заказу определенного потребителя. Такое ПО разрабатывается специально для данного потребителя согласно заключеиному контракту. Программные продукты этого типа включают системы управления для электронных устройств, системы поддержки определенных производственных или бизнес-процессов, системы управления воздушным транспортом и т.п.

Важное отличие между этими типами программных продуктов заключается в том, что при создании общих программных продуктов спецификация требований на них разрабатывается компанией-производителем. Для заказных программных продуктов спецификация обычно разрабатывается организацией, покупающей данный продукт. Спецификация необходима разработчикам ПО для создания любого программного продукта.

1.1.2. Что такое инженерия программного обеспечения

Инженерия программного обеспечения — это инженерная дисциплина, которая охватывает все аспекты создания ПО от начальной стадии разработки системных требований через создание ПО до его использования. В этом определении присутствует две ключевые фразы.

1. "Инженерная дисциплина". Инженеры — это те специалисты, которые выполняют практическую работу. Они применяют теоретические построения, методы и средства там, где это необходимо, но делают это выборочно и всегда пытаются найти решение задачи, даже если не существует подходящей теории или методов решения. Специалисты-инженеры также всегда понимают, что они должны работать в организационных и финансовых рамках заключенных контрактов, т.е. ищут решение поставленной перед иими задачи с учетом условий контракта.
2. "Все аспекты создания программного обеспечения". Инженерия программного обеспечения ие рассматривает *технические* аспекты создания ПО — в ее ведении такие вопросы, как управление проектом создания ПО и разработка средств, методов и теорий, необходимых для создания программных систем.

Можно сказать, что специалисты (инженеры) по программному обеспечению адаптируют существующие методы инженерии ПО к решению своих задач, и зачастую это оказывается наиболее эффективным способом построения высококачественных программных систем. Инженерия программного обеспечения предоставляет всю необходимую информацию для выбора наиболее подходящего метода для множества практических задач. Вместе с тем творческий неформальный подход в определенных обстоятельствах также может быть эффективным. Например, при разработке программных систем электронной коммерции в Internet требуется неформальный подход в сочетании ПО и графического эскизного проектирования.

1.1.3. Различие между инженерией программного обеспечения и компьютерной наукой

Существенное различие заключается в том, что компьютерная наука (computer science) охватывает теорию и методы построения вычислительных и программных систем, тогда как инженерия программного обеспечения акцентирует внимание на практических проблемах разработки ПО. Знание компьютерной науки необходимо специалистам по программному обеспечению так же, как знание физики — инженерам-электронщикам.

В идеале вся инженерия программного обеспечения должна основываться на фундаменте компьютерной науки, но на самом деле это не так. Часто специалисты по программному обеспечению используют подходы, применимые для решения только конкретной задачи (без обобщения на класс подобных задач). Элегантные методы компьютерной науки не всегда можно применить к реальным сложным задачам, требующим программного решения.

1.1.4. Различие между инженерией программного обеспечения и системотехникой

Системотехника (system engineering) или, точнее, технология создания вычислительных систем охватывает все аспекты создания и модернизации сложных вычислительных систем, где программное обеспечение играет ведущую роль. Сюда можно отнести технологию разработки аппаратных средств, внутренних вычислительных процессов и развертывания всей системы, а также технологию создания ПО. Инженеры-системотехники на основе спецификации системы (технических требований) определяют ее архитектуру и затем, собрав воедино ее отдельные части, создают законченную систему. Они рассматривают систему преимущественно как составной объект с заданными компонентами и уделяют сравнительно мало внимания самим системным компонентам (конкретным аппаратным средствам, соответствующему программному обеспечению и т.д.).

Системотехника более старая дисциплина, чем инженерия программного обеспечения. Человечество создает сложные индустриальные системы (такие, как железные дороги и химические заводы) уже более 100 лет. Вместе с тем по мере увеличения в системах роли программного компонента методы инженерии программного обеспечения, например автоматизированное моделирование систем, управление разработкой спецификаций и т.п., все шире используются в процессе создания самых разнообразных систем. Более подробно вопросы системотехники рассматриваются в главе 2.

1.1.5. Процесс создания программного обеспечения

Создание ПО – это совокупность процессов, приводящих к созданию программного продукта. Эти процессы основываются главным образом на технологиях инженерии программного обеспечения. Существует четыре фундаментальных процесса (более подробно описанных далее в книге), которые присущи любому проекту создания ПО.

1. *Разработка спецификации требований на программное обеспечение.* Требования определяют функциональные характеристики системы и обязательны для выполнения.
2. *Создание программного обеспечения.* Разработка и создание ПО согласно спецификации на него.
3. *Аттестация программного обеспечения.* Созданное ПО должно пройти аттестацию для подтверждения соответствия требованиям заказчика.
4. *Совершенствование (модернизация) программного обеспечения.* ПО должно быть таким, чтобы его можно было модернизировать согласно измененным требованиям потребителя.

При выполнении разнообразных программных проектов эти процессы могут быть организованы различными способами и описаны на разных уровнях детализации. Длительность реализации этих процессов также далеко не всегда одинакова. И вообще, различные организации, занимающиеся производством ПО, зачастую используют разные процессы для создания программных продуктов даже одного типа. С другой стороны, определенные процессы более подходят для создания программных продуктов одного типа и менее – для

другого типа программных приложений. Если использовать неподходящий процесс, это может привести к снижению качества и функциональности разрабатываемого программного продукта.

Процессы создания ПО подробно описаны в главе 3, а крайне важная тема усовершенствования технологии создания программных продуктов рассматривается в главе 25.

1.1.6. Модель процесса создания ПО

Такая модель представляет собой упрощенное описание процесса создания ПО – последовательность практических этапов, необходимых для разработки создаваемого программного продукта. Подобные модели, несмотря на их разнообразие, служат абстрактным представлением реального процесса создания ПО. Модели могут отображать процессы, которые являются частью технологического процесса создания ПО, компоненты программных продуктов и действия людей, участвующих в создании ПО. Опишем кратко типы моделей технологического процесса создаиия программного обеспечения.

1. *Модель последовательности работ.* Показывает последовательность этапов, выполняемых в процессе создания ПО, включая начало и завершение каждого этапа, а также зависимость между выполнением этапов. Этапы в этой модели соответствуют определенным работам, выполняемым разработчиками ПО.
2. *Модели потоков данных и процессов.* В них процесс создания ПО представляется в виде множества активностей (процессов), в ходе реализации которых выполняются преобразования определенных данных. Например, на вход активности (процесса) создания спецификации ПО поступают определенные данные, на выходе этой активности получают данные, которые поступают на вход активности, соответствующей проектированию ПО, и т.д. Активность в такой модели часто является процессом более низкого порядка, чем этапы работ в модели предыдущего типа. Преобразования данных при реализации активностей могут выполнять как разработчики ПО, так и компьютеры.
3. *Ролевая модель.* Модель этого типа представляет роли людей, включенных в процесс создания ПО, и действия, выполняемые ими в этих ролях.

Существует также большое количество разнообразных моделей процесса разработки программного обеспечения (т.е. подходов к процессу разработки).

1. *Каскадный подход.* Весь процесс создания ПО разбивается на отдельные этапы: формирование требований к ПО, проектирование и разработка программного продукта, его тестирование и т.д. Переход к следующему этапу осуществляется только после того, как полностью завершаются работы на предыдущем.
2. *Эволюционный подход.* Здесь последовательно перемежаются этапы формирования требований, разработки ПО и его аттестации. Первоначальная программная система быстро разрабатывается на основе некоторых абстрактных (общих) требований. Затем они уточняются и детализируются в соответствии с требованиями заказчика. Далее система дорабатывается и аттестуется в соответствии с новыми уточненными требованиями. Такая последовательность действий может повториться несколько раз.
3. *Формальные преобразования.* Основан на разработке формальной математической спецификации программной системы и преобразовании этой спецификации посредством специальных математических методов в программы. Такое преобразо-

вание удовлетворяет условию "сохранения корректности". Это означает, что полученная программа будет в точности соответствовать разработанной спецификации.

4. *Сборка программного продукта из ранее созданных компонентов.* Предполагается, что отдельные составные части программной системы уже существуют, т.е. созданы ранее. В этом случае технологический процесс создания ПО основное внимание уделяет интеграции отдельных компонентов в общее целое, а не созданию этих компонентов. Эта технология более подробно рассматривается в главе 14.

В главе 3 мы вернемся к моделям технологического процесса создания программного обеспечения.

1.1.7. Структура затрат на создание ПО

Точная структура затрат на создание программного обеспечения существенно зависит от процессов, используемых при разработке ПО, а также от типа разрабатываемого программного продукта. Если принять общую стоимость создания ПО за 100 единиц, то распределение стоимостей отдельных этапов производства может иметь такой вид, как на рис. 1.1.

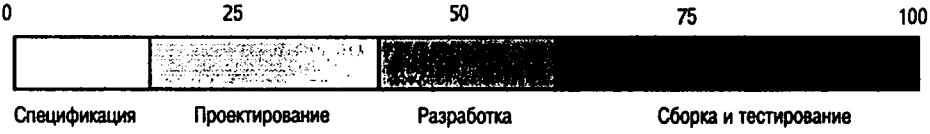


Рис. 1.1. Распределение стоимостей отдельных этапов производства ПО

Примерно такая структура затрат возможна тогда, когда затраты на создание спецификации, проектирование ПО, его разработку и сборку подсчитываются отдельно. Отметим, что часто стоимость этапа сборки и тестирования превышает стоимость этапа непосредственно разработки ПО. Например, на рис. 1.1 показана структура затрат, при которой на тестирование программной системы приходится примерно 40% общей стоимости затрат. Вместе с тем для некоторых критических систем эта статья расходов может превышать 50%.

При использовании эволюционного подхода к разработке ПО практически невозможно провести четкое разграничение между этапами создания спецификации, проектирования и разработки ПО. Поэтому структуру затрат, представленную на рис. 1.1, следует изменить так, как показано на рис. 1.2. Здесь оставлен отдельный этап разработки спецификации, поскольку общая спецификация высшего уровня создается еще до начала создания программного продукта. Создание спецификации нижнего уровня, проектирование, реализация, сборка и тестирование ПО при таком подходе выполняются параллельно на этапе разработки программной системы. Вместе с тем этот подход требует выполнения отдельного этапа тестирования системы после окончания начального этапа ее разработки.

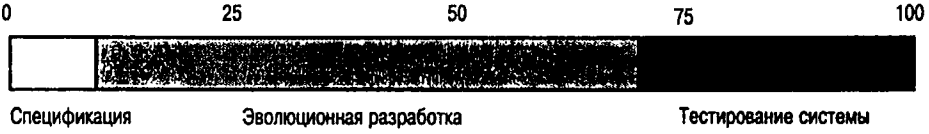


Рис. 1.2. Структура затрат при использовании эволюционного подхода к разработке ПО

В стоимость создания ПО также могут включаться затраты на его модернизацию после начала эксплуатации программного продукта. Для многих программных систем затраты на совершенствование системы могут превышать стоимость разработки в 3 или 4 раза (рис. 1.3).

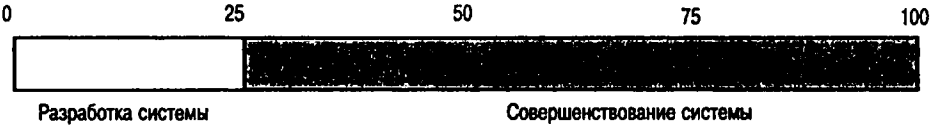


Рис. 1.3. Затраты на разработку и совершенствование ПО

Структура затрат на создание заказного программного обеспечения (т.е. когда требования к системе устанавливаются заказчиком и разработка ПО выполняется по контракту) примерно такая же, как показано выше, но стоимость различных этапов создания программного продукта может значительно разниться. Это относится, в частности, к программам, разрабатываемым для персональных компьютеров. Как правило, такое программное обеспечение разрабатывается на основе эволюционного подхода с использованием уже готового эскиза спецификации. Поэтому стоимость разработки требований к ПО относительно низкая. Вместе с тем такие программные продукты предназначены для работы на разных компьютерных платформах, что существенно повышает затраты на тестирование систем. На рис. 1.4 показана типовая структура затрат на создание такого ПО.

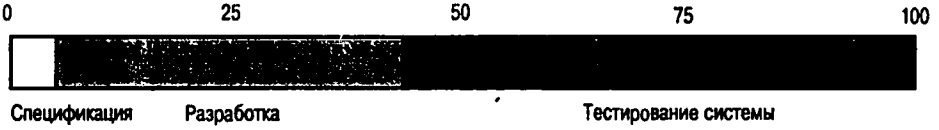


Рис. 1.4. Структура затрат на создание заказного ПО

Стоимость модернизации общих программных продуктов (т.е. тех, которые продаются на открытом рынке программ) с трудом поддается оценке. Во многих случаях осуществляется небольшая формальная модернизация. Обычно с началом реализации созданного программного продукта начинается работа с его следующей версией. Но исходя из требований маркетинга предпочтительнее представить новую версию как новый (но совместимый со старой версией) программный продукт, а не как модифицированную версию того продукта, которую пользователь уже купил. Поэтому стоимость модернизации ПО обычно не подсчитывается отдельно, как это делается при модернизации заказных программных продуктов, а просто входит в стоимость разработки следующей версии программной системы.

Структура затрат на создание систем для электронной коммерции в Internet обычно отличается от того, что описано выше. В таких системах вместо создания программных модулей, управляющих информацией, обычно используют готовое программное обеспечение, а основные затраты приходятся на разработку пользовательских интерфейсов. На момент написания данной книги такие системы только начали разрабатываться и использоваться, поэтому я, к сожалению, не располагаю подходящей схемой, иллюстрирующей структуру затрат на создание такого типа программного обеспечения.

1.1.8. Методы инженерии программного обеспечения

Эти методы представляют собой структурный подход к созданию ПО, который способствует производству высококачественного программного продукта эффективным, в экономическом аспекте, способом. Такие методы, как структурный анализ [91] и JSD (метод Джексона разработки систем) [181], впервые были представлены еще в 1970-х годах. Эти методы, названные функционально-модульными или функционально-ориентированными, связаны с определением основных функциональных компонентов программной системы и в свое время широко использовались. В 80–90-х годах к этим методам до-

бавились объектно-ориентированные методы, предложениые Бучем (Booch) [54] и Рамбо (Rumbaugh) [302]. Эти методы, использующие разные подходы, ныне интегрированы в единый унифицированный метод, построенный на основе унифицированного языка моделирования UML (Unified Modeling Language) [55, 117, 303, 304, 17*, 30*][2].

Все упомянутые методы основаны на идее создания моделей системы, которые можно представить графически, и на использовании этих моделей в качестве спецификации системы или ее структуры. Методы инженерии ПО обычно включают перечисленные в табл. 1.2 компоненты.

Таблица 1.2. Компоненты методов инженерии ПО

Компонент	Описание	Пример
Описание модели системы	Описания моделей создаваемых систем и нотация, используемая для разработки этих моделей	Модели объектов, модели потоков данных, модели конечных автоматов и т.п.
Правила	Правила и ограничения, которые необходимо выполнять при разработке моделей систем	Каждый элемент модели должен иметь уникальное имя
Рекомендации	Эвристические советы и рекомендации, отражающие практический опыт применения данного метода	Любой объект в модели не должен иметь более семи подчиненных ему объектов
Руководство по применению метода	Описание работ, которые необходимо выполнить для построения модели системы, а также рекомендации по организации этих работ	Атрибуты любого объекта должны быть документированы, прежде чем будут определены операции, связанные с этим объектом

Не существует идеального и универсального метода — каждый метод имеет свою область применимости. Например, объектно-ориентированные методы часто применяются для создания интерактивных (диалоговых) программных систем, но практически не используются при разработке систем, работающих в режиме реального времени.

1.1.9. CASE-технология

Аббревиатура CASE обозначает Computer-Aided Software Engineering — автоматизированная разработка программного обеспечения. Под этим понимается широкий спектр программ, применяемых для поддержки и сопровождения различных этапов создания ПО: анализа системных требований, моделирования системы, ее отладки и тестирования и др. Все современные методы создания ПО используют соответствующие CASE-средства: редакторы нотаций, применяемых для описания моделей, модули анализа, проверяющие соответствие модели правилам метода, и генераторы отчетов, помогающие при создании документации на разрабатываемое ПО. Кроме того, CASE-средства могут включать генератор кода, который автоматически генерирует исходный код программ на основе модели системы, а также руководство пользователя.

CASE-средства, предназначенные для анализа спецификаций и проектирования ПО, иногда называют CASE-средствами верхнего уровня, поскольку они применяются на начальной стадии разработки программных систем. В то же время CASE-средства, нацеленные на поддержку разработки и тестирования ПО, т.е. отладчики, системы анализа программ, генераторы тестов и редакторы программ, подчас называют CASE-средствами нижнего уровня.

[2] *Здесь и далее по всей книге звездочкой обозначается литература, добавленная при переводе.* – Прим. ред.

1.1.10. Характеристики качественного программного обеспечения

Кроме функциональных возможностей, присущих программным продуктам по определению, эти продукты обладают и другими показателями, характеризующими их качеств Данные показатели не вытекают непосредственно из того, какие действия может выпо нять программный продукт. Они характеризуют поведение программы во время выпо нения ею своих действий, структуру и организацию исходного кода программы, ее до ментированность. Примером таких показателей (иногда называемых нефункциональны ми показателями) может служить время ожидания пользователем ответа на свой запро или понятность программного кода.

Конечно, множество тех показателей или характеристик, которые можно ожидать о ПО, зависит от типа программной системы. Например, банковская система должна быть з щищенной, интерактивная игра должна быть чувствительной к действиям пользователя-игрока, систему телефонных переключений прежде всего характеризует ее надежность и т. Но эти специфические показатели, как и множество других подобных характеристик, можно обобщить в виде показателей качественных программных систем, приведенных в табл. 1.3.

Таблица 1.3. Основные показатели качественного программного обеспечения

Показатель	Описание
Удобство сопровождения	ПО должно быть таким, чтобы существовала возможность его усовершенствования в ответ на измененные требования заказчика или пользователя. Это определяющий показатель, поскольку любое ПО неминуемо подвергается модернизации вследствие изменений, происходящих в реальном мире
Надежность	Определяется рядом характеристик, таких как безотказность, защищенность и безопасность. Надежность ПО означает, что возможные сбои в работе системы не приведут к физическому или экономическому ущербу
Эффективность	Работа ПО не должна приводить к расточительному расходованию таких системных ресурсов, как память или время занятости процессора. Поэтому эффективность ПО описывается следующими характеристиками: скорость выполнения, используемое процессорное время, объем требуемой памяти и т.п.
Удобство в использовании	ПО должно быть удобным в эксплуатации и не требовать чрезмерного напряжения усилий пользователя того уровня, на которого оно рассчитано. Это означает, что программная система должна обладать соответствующим пользовательским интерфейсом и необходимой документацией

В этой книге основное внимание уделяется только двум из перечисленных показат лей — удобству сопровождения и надежности. Большинство методов, средств и техно гий инженерии программного обеспечения ориентированы на то, чтобы помочь в соз нии программных систем с этими показателями качественного ПО. Освещение темы фективности ПО требует специальных знаний, удобство эксплуатации ПО так отдельная и большая тема, но к ней я еще вернусь в главе 15.

1.1.11. Основные проблемы, стоящие перед специалистами по программному обеспечению

В XXI столетии специалисты по программному обеспечению столкнутся с описанными ниже проблемами.

1. *Проблема наследования ранее созданного ПО.* Многие большие программные системы, эксплуатируемые в настоящее время, созданы миого лет назад, но до сих пор выполняют свои функции надлежащим образом. Проблема наследования означает поддержку и модернизацию таких систем, причем при минимальных финансовых и временны̀х затратах.
2. *Проблема все возрастающей разнородности программны̀х систем.* В настоящее время программное обеспечение должно быть способно работать в качестве систем, распределенных в компьютерных сетях, состоящих из компьютеров разных типов и использующих различные операционные системы. Проблема возрастающей разнородности программных систем состоит в том, что необходимо разрабатывать надежные программные системы, способные работать совместно с ПО разных типов.
3. *Проблема, порожденная требованием уменьшения времени на создание ПО.* Многие традиционные технологии создания качественного программного обеспечения требуют больших временны̀х затрат. Вместе с тем сегодня запросы рынка ПО и требования к программным системам меняются очень быстро. Поэтому и ПО должно меняться с соответствующей скоростью. Проблема, порожденная требованием уменьшения времени на создание ПО, заключается в том, чтобы сократить время на разработку больших и сложных программных систем без снижения их качества.

Конечно, перечисленные проблемы связаны друг с другом. Например, возможна такая ситуация, когда необходимо быстро разработать на основе существующей системы ее сетевой вариант. Для решения таких проблем необходимы новые средства и технологии, которые вобрали бы в себя все лучшие методы современной инженерии программного обеспечения.

1.2. Профессиональные и этические требования к специалистам по программному обеспечению

Подобно любым другим профессионалам, специалисты по программному обеспечению должны согласиться, что к ним предъявляется более широкий круг требований, чем простая необходимость иметь тот или иной профессиональный уровень. Они работают в определенном правовом и социальном окружении. Область инженерии программного обеспечения, как и любая другая сфера человеческой деятельности, имеет ограничения в виде местных, национальных и международных законодательств. Поэтому специалисты по программному обеспечению должны принять на себя определенные этические и моральные обязательства, чтобы стать настоящими профессионалами.

Не требует лишних пояснений утверждение, что специалисты должны быть честными и порядочными людьми. Они не должны использовать свои профессиональные навыки и возможности для деятельности, дискредитирующей профессию специалиста по программному обеспечению. Вместе с тем требования к специалистам не ограничиваются только моральными или юридическими предписаниями, в их круг также входят значительно более тонкие профессиональные обязательства.

1. *Конфиденциальность.* Специалист должен соблюдать конфиденциальность, т.е. не разглашать никаких сведений о работодателе и клиентах, независимо от того, подписывал он или нет какое-либо соглашение о соблюдении конфиденциальности.
2. *Компетентность.* Специалист не должен скрывать (или ложно представлять) свой уровень компетенции и не должен браться за работу, которая этому уровню не соответствует.
3. *Защита прав интеллектуальной собственности.* Специалист не должен нарушать соответствующее законодательство о защите авторских прав при использовании чужой интеллектуальной собственности (патентов и т.п.). Он также должен защищать интеллектуальную собственность работодателя и клиентов.
4. *Злоупотребление компьютером.* Специалист не должен, используя свой профессиональный уровень, наносить вред компьютерам других людей. Злоупотребления компьютером могут быть как относительно тривиальными (скажем, игра в компьютерные игры на машине, принадлежащей работодателю), так и очень серьезными (например, распространение компьютерных вирусов).

В разработке подобных этических обязательств большая роль принадлежит профессиональным обществам и институтам. Такие организации, как ACM (Association for Computing Machinery — Ассоциация по вычислительной технике), IEEE (Institute of Electrical and Electronics Engineers — Институт инженеров по электротехнике и электронике) и British Computer Society (Британское компьютерное общество), опубликовали кодекс профессионального поведения, или этический кодекс. Члены этих организаций принимают на себя обязательство следовать данному кодексу. Правила поведения из этого кодекса основаны на общечеловеческих этических нормах.

ACM и IEEE совместно создали кодекс, соединяющий этические нормы и профессиональную практику. Этот кодекс существует в двух версиях: краткой, приведенной во врезке, и полной [134], раскрывающей, расширяющей и дополняющей основные положения краткой версии кодекса. Обоснование необходимости такого кодекса приведено в первых двух абзацах полной версии.

Вычислительная техника в настоящее время играет все возрастающую роль в деловой сфере, промышленности, медицине, образовании, сфере развлечений и обществе в целом. Инженерия программного обеспечения непосредственно или с помощью своих технологий вносит вклад в анализ и создание спецификации, проектирование, разработку, сертификацию, поддержку и тестирование программных систем. В соответствии со своей ролью в создании программных систем специалисты по программному обеспечению имеют значительные возможности творить добро или делать зло, позволять другим творить добро или делать зло либо влиять на других так, чтобы они творили добро или делали зло. Чтобы быть по возможности уверенным в том, что их усилия направлены только на добро, специалисты по программному обеспечению должны принять на себя обязательство относиться к инженерии программного обеспечения как к общественно полезной и уважаемой профессии. В связи с этим обязательством специалисты по программному обеспечению должны твердо придерживаться следующего Кодекса этики и профессиональной деятельности.

Кодекс содержит восемь принципов поведения и принятия решений специалистами-профессионалами по программному обеспечению, связанных с практической деятельностью, самообучением, управлением, руководством, а также обучением студентов данной профессии. Принципы определяют этику отношений между отдельными личностями, а также в группах и организациях, разделяющих и принимающих эти принципы. Статьи каждого принципа описывают конкретные обязательства, определяющие эти отношения. Истоки этих обязательств – человеческие качества специалистов по про-

граммному обеспечению, особая моральная щепетильность, присущая людям, занимающимся инженерией ПО, и уникальные составляющие практической деятельности специалистов, реализующих методы и технологии инженерии программного обеспечения. Кодекс предлагает обязательства как предписания, предполагающие непременное выполнение, и как высокие цели, к которым должен стремиться специалист по программному обеспечению.

Кодекс этики, разработанный ACM/IEEE (©IEEE/ACM 1999)

Кодекс этики и практической деятельности инженерии программного обеспечения

ACM/IEEE-CS объединили свои усилия для создания Кодекса этики и практической деятельности инженерии программного обеспечения.

ПРЕАМБУЛА

Краткая версия Кодекса резюмирует наши цели в сжатой форме; статьи, которые содержатся в полной версии, дают расширенное и полное толкование наших целей в качестве пути, которым должны следовать профессионалы в области инженерии программного обеспечения. Без учета целей толкования будут скучными юридическими подробностями, без полного толкования цели становятся высоким, но пустым звуком. Вместе цели и толкования образуют целостный кодекс.

Специалисты по программному обеспечению преобразуют выполняемую ими работу по анализу и созданию спецификаций, проектированию и разработке, тестированию и сопровождению программного обеспечения в общественно полезную и уважаемую профессию. В соответствии с этим, кроме общественных обязательств относительно здоровья, безопасности и благополучия общества, специалисты по программному обеспечению должны взять на себя обязательства следовать восьми перечисленным принципам:

1. **Общественные интересы** — деятельность специалистов по программному обеспечению должна проистекать в соответствии с общественными интересами и запросами.
2. **Клиенты и работодатели** — деятельность специалистов по программному обеспечению должна быть направлена на удовлетворение запросов клиентов и работодателей в соответствии с общественными интересами.
3. **Производство** — специалист по программному обеспечению должен гарантировать, что произведенные или модифицированные им программные продукты соответствуют самым высоким, какие только возможны, профессиональным стандартам.
4. **Профессиональные суждения** — специалист по программному обеспечению должен поддерживать честность, непредвзятость и независимость своих профессиональных суждений и оценок.
5. **Управление** — действия руководителей программных проектов должны подчиняться высоким этическим нормам при их руководстве разработкой и сопровождением программного обеспечения.
6. **Профессия** — специалист по программному обеспечению должен поддерживать на высоком уровне репутацию своей профессии в соответствии с общественными интересами.
7. **Коллегиальность** — специалист по программному обеспечению должен поддерживать коллег и быть достойным членом своего коллектива.
8. **Личность** — специалист по программному обеспечению должен постоянно учиться, чтобы соответствовать уровню своей профессии, а также должен руководствоваться высокими этическими нормами в повседневной практической профессиональной деятельности.

Конечно, в одиой и той же ситуации разные люди имеют различные взгляды и принципы решения стоящих перед ними этических проблем. Например, как вы поступите, если не согласны с политикой, которую проводит руководство компании? Очевидно, это зависит от конкретного человека и сути разногласий. Что лучше – дистанцироваться от позиции компании или пересмотреть свои принципы? Если вы считаете, что эти разногласия могут породить проблемы в выполнении программного проекта, будете ли вы

открыто отстаивать свою позицию перед руководством? Если вы рассчитываете работать в этой компании далее, необходимо рано или поздно решать эту нравственную проблему.

Такие этические дилеммы встают перед всеми специалистами в процессе их профессиональной деятельности. К счастью, в большинстве случаев они не имеют существенной принципиальной подоплеки и разрешаются сравнительно просто. Если же не разрешаются, значит, специалист столкнулся, скорее всего, с большой этической проблемой. В принципе ее всегда можно решить, просто уволившись с работы, но в этом случае возможно возникновение других проблем, например с материальным обеспечением семьи.

Особо трудная ситуация для специалиста возникает тогда, когда работодатель ведет себя неэтичным образом. Скажем, компания разрабатывает программную систему, критическую в отношении безопасности. Но вследствие дефицита времени были подделаны протоколы проверки системы на защищенность. Должен ли специалист в этой ситуации поддерживать конфиденциальность, т.е. неразглашение информации о работодателе, либо все же следует предупредить заказчика ПО или придать гласности тот факт, что система может быть незащищенной?

Сложность этой проблемы состоит в том, что не существует критериев абсолютной защищенности систем. Фактическая защищенность системы может быть проверена только в процессе ее длительной эксплуатации. Даже если система удовлетворяет заранее определенным критериям защищенности, это еще не означает, что она лишена ошибок и не может возникнуть каких-либо сбоев в ее работе.

Ранняя постановка рассматриваемой этической проблемы может привести к напряженности между работодателем и служащими, а неблагоприятный исход ее решения может нанести ущерб другим сотрудникам. Вы должны иметь собственную точку зрения на решение возникшей проблемы. Но эта точка зрения обязательно должна учитывать возможные неприятности и ущерб, причиняемый другим людям. Если ситуация очень серьезная, можно, например, обратиться за поддержкой к средствам массовой информации. Но в любом случае нужно пытаться решить проблему без нанесения ущерба правам работодателя.

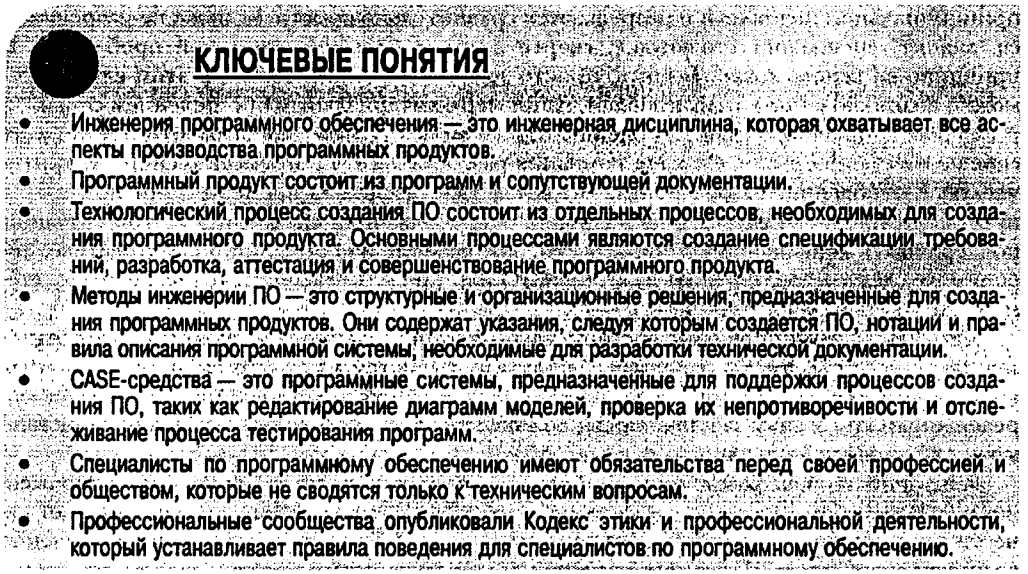
Другая этическая проблема возникает, если вы участвуете в разработке военных или атомных систем. Многие отказываются участвовать в любых разработках, имеющих хоть какое-нибудь отношение к военной тематике. Некоторые согласны работать над военной тематикой, не связанной с производством вооружения, а кто-то считает, что национальная безопасность — достаточное основание, чтобы не иметь этических проблем при работе над системами вооружений. Здесь соответствующая этическая позиция полностью зависит от взглядов и мировоззрения конкретного человека.

В этой ситуации важно, чтобы как работодатель, так и работники заблаговременно узнали взгляды друг друга. Когда организация включается в военные или атомные проекты, руководство должно сообщить коллективу, что они должны быть готовыми принять любую работу соответствующего профиля. С другой стороны, если штатные сотрудники не хотят принимать участие в разработке военных систем, руководство не должно оказывать на них давление, принуждая к выполнению таких работ.

К вопросам профессиональных этических норм и отношений интерес возрастает на протяжении ряда последних лет. Их можно рассматривать исходя из философских категорий, описывающих базовые принципы этики, и на основе этики инженерии программного обеспечения, которая также ссылается на эти принципы. Такой подход представлен в работе [210] и в меньшей степени в [164].

Однако я нахожу такой подход слишком абстрактным и трудным для "привязки" к моей повседневной практике. Я отдаю предпочтение более практичному подходу, реализованному в кодексах поведения и профессиональной деятельности. Я думаю, что этические проблемы лучше обсуждать в практическом контексте инженерии программного обеспе-

чения, а не как субъекты абстрактных общих прав человека. Поэтому в данной книге я не касаюсь абстрактных этических дискуссий, но там, где это возможно, привожу примеры из реальной жизни, которые могут служить основой для дальнейшего обсуждения.



КЛЮЧЕВЫЕ ПОНЯТИЯ

- Инженерия программного обеспечения — это инженерная дисциплина, которая охватывает все аспекты производства программных продуктов.
- Программный продукт состоит из программ и сопутствующей документации.
- Технологический процесс создания ПО состоит из отдельных процессов, необходимых для создания программного продукта. Основными процессами являются создание спецификации требований, разработка, аттестация и совершенствование программного продукта.
- Методы инженерии ПО — это структурные и организационные решения, предназначенные для создания программных продуктов. Они содержат указания, следуя которым создается ПО, нотаций и правила описания программной системы, необходимые для разработки технической документации.
- CASE-средства — это программные системы, предназначенные для поддержки процессов создания ПО, таких как редактирование диаграмм моделей, проверка их непротиворечивости и отслеживание процесса тестирования программ.
- Специалисты по программному обеспечению имеют обязательства перед своей профессией и обществом, которые не сводятся только к техническим вопросам.
- Профессиональные сообщества опубликовали Кодекс этики и профессиональной деятельности, который устанавливает правила поведения для специалистов по программному обеспечению.

Упражнения

1.1. На основе схем структуры затрат на создание ПО, представленных в разделе 1.1.7, объясните, почему затраты на первоначальное обдумывание и обсуждение создаваемой программной системы могут превосходить стоимость продаваемых программ.

1.2. Назовите четыре основные характеристики, которыми должен обладать любой программный продукт. Предложите четыре другие характеристики, которые также существенны для программных систем.

1.3. Каково различие между моделью процесса создания ПО и самим процессом? Приведите две ситуации, когда модель процесса создания ПО может быть полезной в определении возможных этапов совершенствования программного продукта.

1.4. Методы инженерии программного обеспечения широко используют CASE-технологии для поддержки процесса создания ПО. Назовите пять этапов процесса разработки ПО, где находят применение CASE-средства.

1.5. Помимо проблем наследования ранее созданного ПО, возрастающей разнородности программных систем и проблемы, порожденной требованием уменьшения времени на создание ПО, назовите другие проблемы, также стоящие перед инженерией ПО.

1.6. Обсудите вопрос о том, должны ли специалисты по программному обеспечению иметь соответствующие сертификаты, как, например, врачи и юристы.

1.7. Для каждой статьи Кодекса этики и профессиональной деятельности приведите соответствующие примеры, иллюстрирующие эти статьи.

2

Системотехника вычислительных систем

Цели

Цель настоящей главы — дать введение в системотехнику вычислительных систем и объяснить, почему знания в этой области важны для специалистов по программному обеспечению. Прочитав эту главу, вы должны:

- понимать, почему программное обеспечение все шире применяется в разработках системотехники;
- знать основные интеграционные характеристики систем: безотказность, производительность, надежность и безопасность;
- понимать, почему в процессе разработки систем необходимо учитывать окружение, в котором будет работать система;
- иметь представление о процессах создания систем и системного обеспечения.

Содержание

Системотехника, как технология создания систем, охватывает процессы создания спецификаций, проектирования, разработки, тестирования, внедрения и сопровождения систем *как единого целого*. Системотехник, занимающийся разработкой вычислительных систем, не сосредоточен только на программном обеспечении, он уделяет равное внимание программному обеспечению, аппаратным средствам и средствам взаимодействия с пользователями и системным окружением. Он должен думать о тех функциях, которые будет выполнять система и, собственно, ради которых строится система, а также о взаимодействии системы с ее окружением. Специалист по созданию программного обеспечения должен понимать задачи системотехники, поскольку возникающие проблемы часто являются результатом решений, принятых системотехниками.

Существует множество самых разнообразных определений понятия "система", от очень абстрактных до чрезвычайно конкретных. С моей точки зрения, наиболее удачным определением системы будет следующее.

Система – это совокупность взаимодействующих компонентов, работающих совместно для достижения определенных целей.

Это определение охватывает очень широкий круг систем. Например, такая простая система, как карандаш, состоит из двух или нескольких "аппаратных" компонентов, в то время как система управления полетами может состоять из тысяч аппаратных и программных компонентов плюс операторы, которые принимают решения на основе данных, предоставляемых информационными подсистемами.

Определяющим признаком системы является то, что свойства и поведение системных компонентов влияют друг на друга чрезвычайно сложным и запутанным образом. Корректное функционирование каждого системного компонента зависит от функционирования многих других компонентов. Так, операционное обеспечение может выполнять свои функции, если только работает процессор и соответствующие аппаратные средства. Процессор может выполнять вычисления, если только корректно инсталлированы программные системы, задающие эти вычисления.

Системы часто имеют иерархическую структуру, т.е. в качестве компонентов содержат другие системы. Например, компьютерная система полиции содержит географическую информационную систему, которая предлагает информацию о месте, где случилось или может случиться какое-либо происшествие. Системы, которые являются компонентами других систем, называются *подсистемами*. Определяющее свойство подсистем заключается в том, что они могут функционировать самостоятельно, независимо от тех систем, в состав которых входят. Поэтому географическую информационную систему можно использовать также в других системах. Вместе с тем их поведение в составе какой-либо конкретной системы зависит, конечно, от взаимодействия с другими подсистемами.

Сложность взаимодействия между системными компонентами означает, что система не сводится просто к сумме ее составных частей. Она имеет определенные свойства, которые присущи ей именно как целостной системе. Такие *интеграционные свойства* не могут быть свойствами какой-либо отдельной части системы. Более того, они проявляются тогда, когда система рассматривается как единое целое. Некоторые из этих свойств можно вывести из аналогичных свойств отдельных подсистем, но чаще они являются комплексным результатом взаимодействия всех подсистем и их невозможно оценить, исходя из анализа отдельных системных компонентов.

Приведем примеры интеграционных свойств.

1. *Суммарный размер системы.* Это пример интеграционного показателя системы, который можно вычислить, исходя только из свойств отдельных компонентов.

2. *Безотказность системы.* Это свойство зависит от безотказности отдельных компонентов и взаимосвязи между ними.
3. *Удобство эксплуатации системы.* Это очень сложное многопараметрическое свойство, которое зависит не только от программного обеспечения и аппаратных средств системы, но также от окружения, в котором эксплуатируется система, и от системных операторов.

В этой книге основное внимание уделяется вычислительным системам, которые состоят из аппаратных и программных компонентов и, как правило, имеют подсистемы для взаимодействия с человеком. Специалист по программному обеспечению должен знать системотехнику вычислительных систем, поскольку здесь программный компонент играет очень важную роль. Например, в 1969 году для осуществления посадки человека на Луна в программе "Аполлон" использовалось ПО, занимающее всего 10 Мбайт, а ПО космической станции – 100 Мбайт. Таким образом, технологии инженерии программного обеспечения часто являются критическим фактором при разработке сложных вычислительных систем.

2.1. Интеграционные свойства систем

Как отмечалось выше, интеграционные свойства систем проявляются только тогда, когда система рассматривается как единое целое. В этом состоит сложность прогнозирования и оценки таких свойств, поскольку иногда можно измерить показатели только подсистем, из которых состоит комплексиая система.

Существует два типа интеграционных свойств.

1. Функциональные свойства, которые проявляются только тогда, когда система работает как единое целое. Например, велосипед имеет функциональные свойства транспортиого средства только тогда, когда собран из своих компонентов.
2. Нефункциональные свойства: безотказность, производительность, безопасность и защищенность (ограничение несанкционированного доступа к системе), которые зависят от поведения системы в операционном окружении. Такие свойства часто критичны для вычислительных систем, поскольку если они не достигают определенного миимального уровня, то система ие будет работоспособной. Некоторые функции и возможности системы могут быть не востребованы всеми пользователями, так что система может быть работоспособной и без них. Вместе с тем система, не надежная или не эффективная в своих отдельных функциях, все равно считается бракованной.

Чтобы проиллюстрировать сложность в определении интеграционных свойств, рассмотрим такой показатель системы, как безотказность. Это комплексный показатель, который всегда следует рассматривать на уровне системы, а не ее отдельных компонентов. Компоненты в системе взаимосвязаны, так что сбой в одном компоненте может распространиться по всей системе и вызвать ответную реакцию в других компонентах. Проектировщики систем часто не могут предугадать последовательность распространения сбоев в системе, поэтому трудно оценить безотказность системы только на основании данных о безотказности ее отдельных компонентов.

Существует три тесно связанных между собой фактора, которые влияют на общую безотказность системы.

1. *Безотказность аппаратных средств.* Этот показатель определяется вероятностью выхода из строя отдельных аппаратных компонентов и временем, необходимым на их замену.

2. *Безотказность программного обеспечения.* Это показатель работы компонента ПО без сбоев и ошибок. Программные ошибки обычно не оказывают влияния на аппаратные средства системы. Поэтому система может продолжать функционировать даже тогда, когда ПО выдает некорректные результаты. Безотказность программного обеспечения подробно рассматривается в главах 16 и 17.
3. *Ошибки операторов.* Операторы, эксплуатирующие систему, также могут допускать ошибки в своей деятельности.

Все перечисленные факторы тесно связаны между собой. Сбои в аппаратных средствах могут породить ложные сигналы, которые затем поступают на вход программных компонентов, что, в свою очередь, может привести к непредсказуемому поведению программного обеспечения. Операторы обычно допускают ошибки в нештатных ситуациях, когда система ведет себя необычным образом. Такие ситуации часто порождаются какими-либо сбоями в системе. Неправильные действия оператора, в свою очередь, могут спровоцировать сбои и ошибки в работе аппаратных средств, что также может привести к дальнейшему распространению сбойных и ложных сигналов по системным цепям. Таким образом, небольшая ошибка, возникшая в одной подсистеме и в принципе легко устранимая, может привести к ситуации, требующей полного отключения системы.

Безотказность системы также зависит от окружения, в котором она эксплуатируется. Как указывалось выше, трудно предвидеть системное окружение, в котором будет эксплуатироваться система. Другими словами, сложно описать окружение в виде ограничений, которые должны учитываться при разработке системы. Подсистемы, составляющие целостную систему, могут по-разному реагировать на изменения в системном окружении, тем самым влияя на общую безотказность системы самым непредвиденным образом. Вследствие этого, даже если система является единым целым, бывает трудно или совсем невозможно измерить уровень ее безотказности.

Допустим, система предназначена для эксплуатации при нормальной комнатной температуре. Для того чтобы система могла функционировать при других температурных режимах, ее электронные компоненты должны быть рассчитаны для работы в определенном температурном интервале, скажем, от 0 до 45°. При выходе из этого температурного интервала компоненты могут вести себя непредсказуемым образом. Теперь предположим, что система является внутренней составной частью воздушного кондиционера. Если кондиционер неисправен и гонит горячий воздух через электронные компоненты, то они, а следовательно, и вся система могут выйти из строя. Если кондиционер работает нормально, то система также должна работать нормально. Но вследствие физической замкнутости кондиционера могут возникнуть непредвиденные влияния разных компонентов устройства друг на друга, что также может привести к различным сбоям.

Подобно безотказности, другие интеграционные характеристики (такие, как производительность и удобство эксплуатации) также трудны для определения, но могут быть оценены в процессе эксплуатации системы. Оценка других свойств, например безопасности системы и ее защищенности, порождает большие сложности. Эти свойства не просто присущи работающей системе, они отражают те характеристики, которые она *не показывает*. Например, при разработке мер защищенности, где одним из показателей является невозможность несанкционированного доступа к данным, сравнительно легко просчитать все возможные режимы доступа к данным и исключить нежелаемые. Поэтому оценить уровень защищенности можно только через характеристики системы, присущие ей по умолчанию. Более того, система будет считаться обладающей свойством защищенности до тех пор, пока кто-нибудь не взломает ее средства защиты.

2.2. Система и ее окружение

Любая система зависит от сигналов, данных или другой информации, поступающей на ее входы; иными словами, система функционирует в определенном окружении, которое влияет на ее функционирование и производительность. Иногда окружение можно рассматривать как самостоятельную систему, состоящую из множества других систем, которые влияют друг на друга.

На рис. 2.1 показано несколько систем, объединенных в систему жизнеобеспечения офисного здания. Система отопления, электроэнергетическая система, система освещения, системы водоснабжения и канализации и система безопасности являются подсистемами строения, которое, в свою очередь, также можно рассматривать как систему. Здание расположено на улице, которая также является системой более высокого уровня. Улица будет подсистемой системы города и т.д. Таким образом, окружение какой-либо системы само является системой более высокого уровня. В общем случае окружение какой-либо системы – это композиция ее локального окружения и окружения системы более высокого уровня.

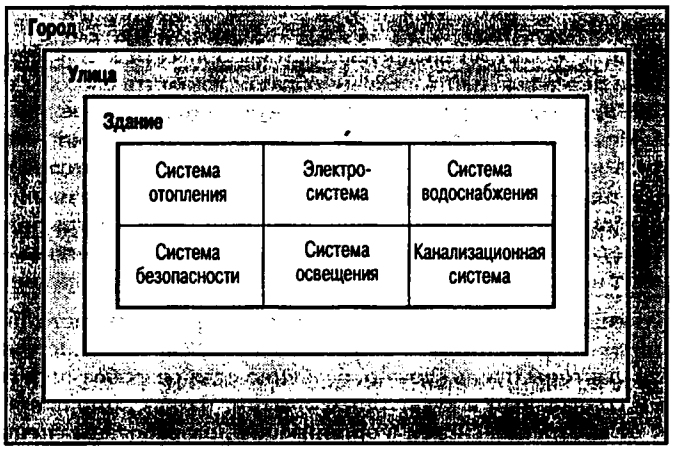


Рис. 2.1. Иерархия систем

Рассмотрим систему безопасности, входящую в систему жизнеобеспечения здания (см. рис. 2.1). Ее локальное окружение состоит из других систем этого здания. К окружению системы также необходимо отнести системы, не входящие в систему здания, но относящиеся к системе улицы и системе города, включая естественные природные системы, в том числе погодную (т.е. воздействие погодных факторов на систему безопасности).

Приведем две основные причины, которыми вызвана необходимость учитывать при разработке систем их окружение.

1. Во многих случаях система предназначена как раз для реагирования на изменение определенных параметров окружения. Так, система отопления реагирует на изменения в окружающей среде, повышая или понижая температуру своих отопительных приборов. Здесь правильное функционирование системы проявляется именно как реакция на изменения параметров окружения.
2. Часто качество функционирования системы может зависеть от параметров окружения самым непредсказуемым образом. Так, система электроснабжения напрямую зависит от уличного окружения здания. Например работы, проводимые по благоустройству улицы, по недосмотру могут повредить силовой кабель и, следовательно,

вывести из строя всю систему электроснабжения здания. Либо грозовой разряд может индуцировать большие токи в электрической системе, что может нарушить ее нормальное функционирование.

Кроме физического окружения (окружающей среды), показанного на рис. 2.1, системы могут находиться в определенных отношениях с организационным окружением, которое включает в себя правила и процедуры, основанные на политических, экономических и экологических приоритетах общества. Если система построена без учета организационного окружения, она может не найти спроса на рынке системных продуктов и будет отвергнута пользователями и потенциальными потребителями.

На разработку систем влияют как человеческие, так и организационные факторы, входящие в окружение системы.

1. *Эксплуатационный фактор.* Требует ли система внесения изменений в рабочий процесс ее эксплуатации, в зависимости от изменения параметров окружения? Если ответ на этот вопрос положительный, следовательно, необходимо обучение персонала, эксплуатирующего эту систему. Если обучение длительное или персонал может потерять в заработке, существует вероятность, что такая система будет отвергнута пользователями.
2. *Фактор персонала.* Может ли внедрение системы привести к снижению требований к квалификации персонала или коренным образом изменить способы его работы? Если это так, тогда персонал может попытаться противостоять внедрению системы в их организацию. Менеджеры среднего звена, руководящие проектами, часто подозревают, что их статус в организации понизится после внедрения компьютерных систем.
3. *Организационный фактор.* Иногда внедрение системы может изменить структуру властных полномочий в организации. Например, если деятельность организации напрямую зависит от правильного функционирования сложной системы, операторы этой системы могут иметь значительный вес во властной структуре организации.

Эти человеческие, социальные и организационные факторы часто оказываются критическими при принятии решения о том, будет или нет внедряться система. К сожалению, предусмотреть эти факторы очень сложно, особенно если разработчики системы не обладают достаточным социальным и культурным опытом. Чтобы помочь предусмотреть различные эффекты от внедрения систем в организацию, разработаны специальные методологии, такие как социотехника Мамфорда (Mumford) [243], методология программных систем Чекланда (Checkland) [69, 70]. Углубленное социологическое исследование эффектов внедрения вычислительных систем приведено в работе [3].

В идеале все сведения о системном окружении следует включить в спецификацию системы с тем, чтобы разработчики могли их учесть при ее проектировании. Но в реальной действительности это невозможно. Обычно разработчики систем делают предположения о системном окружении либо на основе опыта эксплуатации других подобных систем, либо исходя из здравого смысла. Если они ошибутся, то система может в некоторых ситуациях функционировать некорректно. Например, если разработчики не учтут возможные электромагнитные наводки на систему, то она может выйти из строя, если вблизи ее располагаются другие системы с большим электромагнитным излучением.

2.3. Моделирование систем

В процессе формализации требований к системе и на этапе проектирования система рассматривается как совокупность компонентов и взаимосвязей между ними. Для этого используются модели системной архитектуры, которые в графическом виде предоставляют всю организацию системы, т.е. ее компоненты и взаимосвязи между ними.

Архитектура системы обычно представляется в виде блочной диаграммы (блок-схемы), где блоки соответствуют основным подсистемам, а существующие связи между подсистемами обозначаются линиями со стрелками, соединяющими отдельные блоки диаграммы. Связи могут соответствовать потокам данных, последовательности включения подсистем в работу или каким-либо другим типам зависимости.

На рис. 2.2 представлена блок-схема основных компонентов системы сигнализации, предупреждающей о несанкционированном проникновении в жилище. В табл. 2.1 приведено краткое описание подсистем, которым соответствуют определенные блоки на рис. 2.2.

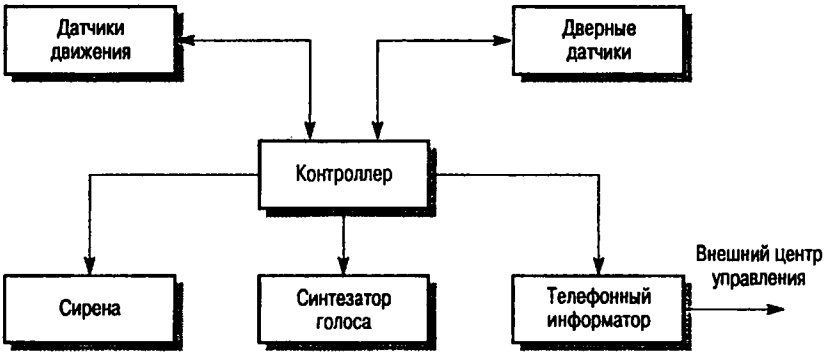


Рис. 2.2. Простая система сигнализации

Таблица 2.1. Функциональные подсистемы системы сигнализации

Подсистема	Описание
Датчики движения	Реагируют на движение в комнатах, которые контролирует система
Дверные датчики	Определяют, открыты ли наружные двери дома
Контроллер	Управляет действиями всей системы
Сирена	Издает мощный звуковой сигнал при незаконном проникновении в жилище
Синтезатор голоса	Синтезирует голосовое сообщение о проникновении в дом
Телефонный информатор	Делает внешний телефонный звонок для уведомления службы безопасности (например, полиции) о проникновении в дом

На этом уровне детализации система разбивается на отдельные подсистемы. Каждая подсистема, в свою очередь, может быть представлена как декомпозиция своих функциональных компонентов. Это такие компоненты подсистемы, которые, исходя из предназначения подсистемы, выполняют какую-либо одну функцию. В противоположность этому подсистема обычно выполняет несколько функций. Конечно, декомпозицию подсистем (и самой системы) можно проводить по другим признакам, например конструктивным или технологическим.

Исторически сложилось так, что модель системной архитектуры используется для вычленения аппаратных и программных компонентов системы, которые обычно разрабатываются параллельно. Вместе с тем противопоставление "аппаратные средства — программное обеспечение" в современных системах чаще всего неуместно и несущественно, поскольку практически все системные компоненты обладают определенными вычисли-

тельными возможностями. Например, машины, связывающие множество компьютеров в единую сеть, состоят из репитеров[1], сетевых шлюзов[2] и соединительных кабелей. Репитеры и шлюзы имеют процессоры и программы, управляющие этими устройствами, и, конечно же, другие электронные компоненты.

На уровне системной архитектуры более рационально классифицировать подсистемы в соответствии с выполняемыми ими функциями, не акцентируя специально внимание на том, являются ли они аппаратными или программными компонентами. Вопрос о том, будет ли данная функция реализована аппаратно или программно, часто решается на основе нетехнических факторов, таких как время, необходимое для создания компонента, или исходя из наличия на рынке промышленных изделий подходящих готовых устройств.

Блок-схемы можно использовать для представления систем любого размера. На рис. 2.3 показана архитектура значительно более сложной системы управления полетами. Эта система содержит несколько основных подсистем, которые сами являются системами большого размера. Направление информационных потоков между подсистемами показано соединяющими их линиями со стрелками.

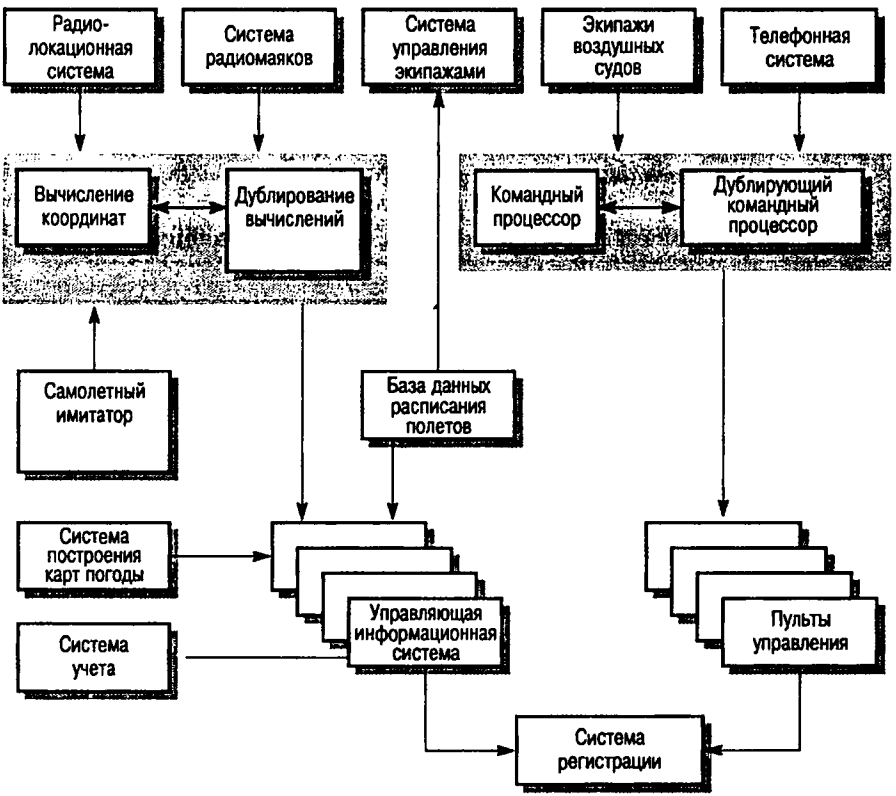


Рис. 2.3. Архитектура системы управления полетами

[1] *Репитер – устройство, передающее сигналы, поступающие по одному кабелю, в другой кабель без маршрутизации или фильтрации пакетов.* – Прим. ред.

[2] *Шлюз – устройство для объединения сетей, использующих различные протоколы передачи пакетов.* – Прим. ред.

2.3.1. Функциональные компоненты систем

Как отмечалось в предыдущем разделе, системная архитектура описывается в терминах функциональных подсистем, независимо от того, являются ли эти подсистемы аппаратными или программными. Вместе с тем функциональные компоненты в системе можно классифицировать по целому ряду категорий, некоторые из них приведены ниже.

1. *Сенсорные компоненты* собирают информацию о системном окружении. Примерами могут служить радиолокаторы в системе управления полетами, датчик положения бумаги в лазерном принтере или термопара в топочной камере котла.
2. *Исполнительные компоненты* производят некоторые действия в окружении системы. Примерами могут служить регулирующий клапан, закрывающий или отрывающий заслонку в трубопроводе для уменьшения или увеличения скорости потока жидкости в нем, закрылки крыльев самолета, которые управляют углом наклона самолета, механизм подачи бумаги в лазерном принтере.
3. *Вычислительные компоненты* — на их вход поступают определенные данные, в соответствии с которыми они выполняют вычисления, затем на выходе получают новые данные. Примером вычислительного компонента является математический сопроцессор, выполняющий вычисления с числами в экспоненциальном формате.
4. *Коммуникационные компоненты* предоставляют возможность другим системным компонентам обмениваться информацией. В качестве примера назовем сетевые интерфейсные платы компьютеров, объединенных в локальную сеть.
5. *Координирующие компоненты* согласуют работу других компонентов. Примером является планировщик заданий в системах реального времени. Планировщик определяет, какой процесс в данный момент времени может обрабатываться процессором.
6. *Интерфейсные компоненты* преобразуют систему представлений, которыми оперирует один системный компонент, в систему представлений, применяемых другим компонентом. Примером "человеческого" интерфейсного компонента может служить модель какой-нибудь системы и представление ее в виде, понятном другому человеку. Другим примером является аналогово-цифровой преобразователь, преобразующий аналоговый сигнал в последовательность чисел.

В табл. 2.2 описан тип функциональных компонентов архитектуры системы сигнализации, представленной на рис. 2.2.

Таблица 2.2. Типы компонентов системы сигнализации

Тип компонента	Компонент	Функции компонента
Сенсорный	Датчик движения, дверной датчик	Регистрирует движение в защищенном помещении, определяет, открыта ли наружная дверь
Исполнительный	Сирена	Издает звуковой сигнал при незаконном проникновении в жилище
Коммуникационный	Телефонный информатор	Делает телефонный звонок в центр управления при проникновении в дом. Получает ответную команду из центра управления
Координирующий	Контроллер	Координирует все системные компоненты. Действует по командам панели управления и центра управления
Интерфейсный	Синтезатор голоса	Синтезирует сообщение о проникновении в дом

Конечно, несложно отнести системные компоненты к одному из перечисленных типов. Вместе с тем, если в системе используется программное обеспечение, то, как правило, программные элементы встраиваются в большинство системных компонентов. Программное обеспечение обычно используется для управления всей системой.

Приведенная классификация компонентов помогает при проектировании систем. Большинство систем содержат компоненты всех типов, и задача разработчика состоит в точном определении типа компонента исходя из спецификации системы. Если несколько компонентов содержат признаки разных типов, это может привести к тому, что при проектировании системы могут возникнуть определенные проблемы.

2.4. Процесс создания систем

Этапы процесса создания системы показаны на рис. 2.4. Эти этапы оказывают большое влияние на процесс разработки программного обеспечения в соответствии с каскадной моделью, которая описывается в главе 3.

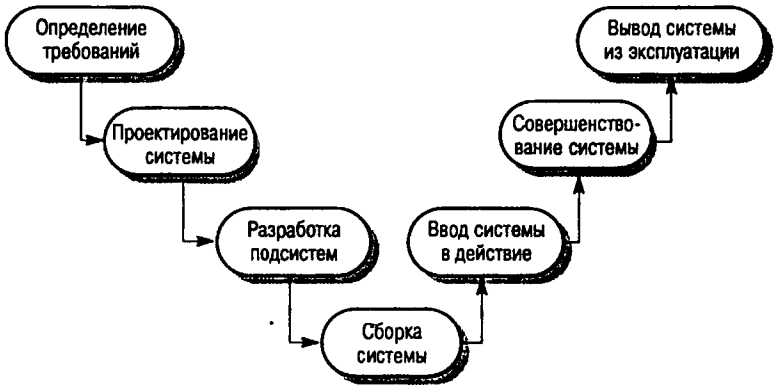


Рис. 2.4. Процесс создания системы

Опишем основные отличия между процессом создания систем и процессом разработки программного обеспечения.

1. *Вовлечение в процесс разработки систем разнообразных инженерных дисциплин.* Процесс создания систем обычно требует привлечения разнообразных инженерных дисциплин. Это может привести к значительным затруднениям в разработке систем, поскольку каждая дисциплина использует свою терминологию.
2. *Небольшой масштаб повторных работ при разработке систем.* После принятия решений в процессе разработки систем (например, об установке определенных типов радиолокаторов в системе управления полетами) внесение изменений в систему может оказаться весьма дорогостоящим. Перепроектирование системы часто просто невозможно. Это одна из причин широкого использования ПО при создании самых разнообразных систем – программные компоненты делают системы более гибкими и позволяют внести изменения в разрабатываемую систему в ответ на новые требования, предъявляемые к ней.

В команду разработчиков систем неизбежно включаются специалисты разных профилей. Команда разработчиков должна обладать широким кругом знаний, чтобы всесторонне рассмотреть все системные возможности при принятии каких-либо решений. Рассмот-

рим систему управления полетом (СУП), в которой используется радиолокационная или какая-нибудь другая сенсорная система для определения местонахождения самолетов (см. рис. 2.3). На рис. 2.5 схематично показаны те инженерные дисциплины, которыми должны владеть члены команды разработчиков системы.

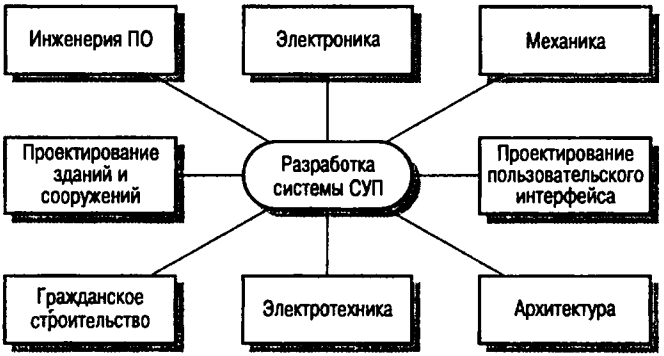


Рис. 2.5. Инженерные дисциплины, вовлекаемые в процесс системотехники

Для многих систем существует практически бесконечное количество способов декомпозиции (разбиения) системы на подсистемы. При этом специалисты разных профилей могут предлагать различные варианты структурной модели системы, которые будут содержать разные функциональные компоненты. Тем самым возможен широчайший диапазон альтернативных моделей. Выбор определенной модели не обязательно основывается только на технических аргументах. Пусть, например, одной из альтернатив в разработке СУП является установка новой радиолокационной системы вместо модернизации существующей. Если в команду разработчиков входят строители, то они могут настоять именно на этом варианте создания СУП, так как он обеспечит работой и их, и строительные подразделения, которые они представляют. При этом для обоснования нужного варианта могут, конечно, привлекаться и технические аргументы.

Поскольку ПО по своей природе является гибким и сравнительно легко настраиваемым, часто решение многих неожиданно возникших проблем перекладывается на плечи специалистов по программному обеспечению. Пусть, например, при создании СУП неудачно выбрано местоположение одной радарной установки — на экране локатора иногда происходит раздвоение изображений. Для удаления этого эффекта необходимо передвинуть радарную установку, что практически не осуществимо. Решением этой проблемы может быть создание специального ПО, которое поможет удалить раздвоение изображений. Но в этом случае может потребоваться более мощная вычислительная техника, чем та, которая изначально запланирована, и это, в свою очередь, также может стать определенной проблемой.

Перед специалистами по программному обеспечению часто ставятся задачи, которые необходимо решать без увеличения стоимости аппаратных средств. Поэтому многие так называемые программные ошибки не являются следствием каких-либо "врожденных" черт или свойств ПО. Они могут быть результатом попытки модернизировать программное обеспечение в соответствии с изменениями требований, предъявляемых к создаваемой системе. Хорошим примером такой ошибки может служить сбой в системе управления багажом в аэропорту Денвера [329].

2.4.1. Определение системных требований

На этапе определения системных требований формируются и формализуются требования к системе, рассматриваемой как единое целое. Как и при анализе требований к программному обеспечению, здесь также необходимы консультации с заказчиками системы и ее конечными пользователями. На этапе определения требований обычно формируются требования трех типов.

1. *Общие функциональные требования.* Основные функции, выполняемые системой, определяются на самом высшем (абстрактном) уровне представления системы. Детализация функциональных требований происходит уже на уровне подсистем. Например, при разработке СУП обязательно будет предусмотрено требование иметь базу данных полетов, совершенных в контролируемом системой воздушном пространстве. Однако структура этой базы данных не будет определена до тех пор, пока не будут отработаны требования к другим подсистемам.
2. *Системные свойства.* Это те интегрированные свойства системы, которые обсуждались выше. Они могут включать такие свойства, как производительность, безотказность, защищенность и т.п. Эти нефункциональные свойства оказывают влияние на все требования, определяемые для подсистем.
3. *Свойства, которые должны отсутствовать у системы.* Порой гораздо важнее указать, что система не должна делать, чем то, что она должна выполнять. Например, в СУП необходимо потребовать, чтобы система не предоставляла операторам слишком много информации, только самую необходимую, не отвлекающую их внимание.

Важной частью этапа определения требований является описание множества целей, к выполнению которых должна стремиться система. Они не обязательно должны быть выражены в терминах функциональных свойств системы, но должны показать, как она будет себя вести в своем окружении.

Чтобы проиллюстрировать описание множества целей, рассмотрим объединенную систему противопожарной безопасности и защиты от несанкционированного вторжения, предназначенную для установки в офисном здании. Цели, которые основываются на функциональных возможностях системы, можно сформулировать следующим образом.

Система должна обеспечить предупреждения о возгораниях, возникших внутри или вблизи здания, и несанкционированном проникновении в это здание.

Эта цель точно описывает назначение системы, которая должна предупреждать о неких нежелательных событиях. Такая формулировка подходит для системы безопасности, которая уже существует и которая должна быть заменена. В противоположность этому можно сформулировать более "широкую" цель.

Система должна гарантировать отсутствие серьезных нарушений в нормальном функционировании и эксплуатации здания вследствие возгораний и незаконных вторжений.

Первая формулировка цели ограничивает возможности проектирования системы. В соответствии с ней от несанкционированных проникновений можно применить сложные защитные средства даже без внутренней системы сигнализации, а для защиты от огня можно использовать автоматическую систему пожаротушения с разбрызгивателями воды. Но такие средства могут вывести из строя электрическую систему и причинить серьезные неудобства работающим в здании.

Подчас основная трудность в определении системных требований состоит в том, что система строится для того, чтобы помочь в решении "злостной" проблемы (wicked

problem) [294]. "Злостная" проблема – это проблема такой большой сложности и имеющая столько взаимосвязанных входных воздействий, что ее невозможно точно описать. Истинная природа такой проблемы может проявиться только в процессе ее решения. В качестве экстремального примера "злостной" проблемы можно привести задачу предсказания землетрясений. В настоящее время не существует точных способов предсказания ни эпицентра землетрясения, ни его времени, ни силы, ни воздействия на окружающую среду. Поэтому невозможно заранее полностью спланировать все действия на случай большого землетрясения – это можно сделать только тогда, когда оно произойдет.

2.4.2. Проектирование систем

Проектирование системы (рис. 2.6) заключается в определении системных компонентов на основе функциональных требований к системе. Процесс проектирования состоит из нескольких этапов.

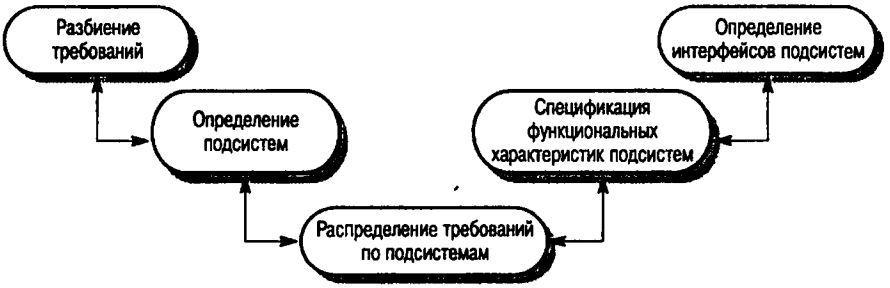


Рис. 2.6. Этапы проектирования системы

1. *Разбиение требований на группы.* Требования анализируются и разбиваются на отдельные группы. Обычно множество требований можно разбить на группы многими способами, на этом этапе сохраняются все "жизнеспособные" разбиения.
2. *Определение подсистем.* Определяются подсистемы, которые индивидуально или совместно реализуют системные требования. Группа требований обычно проецируется на несколько подсистем, поэтому можно объединить несколько требований в одно. Вместе с тем на определение систем влияют не только системные требования, но и организационные или производственные факторы.
3. *Распределение требований по подсистемам.* В принципе эта операция должна быть выполнена на предыдущем этапе определения подсистем. Но на практике не всегда можно четко согласовать разбиение требований и определение подсистем. Или, например, ограниченный ассортимент подсистем, приобретаемых у внешних поставщиков (см. раздел 2.4.3), может привести к пересмотру требований к системе.
4. *Специфицирование функциональных характеристик подсистем.* Определяются функциональные характеристики каждой подсистемы. Если подсистема является программной, этот этап будет частью этапа создания спецификации для данной подсистемы. На этом этапе также формализуются взаимоотношения между подсистемами.
5. *Определение интерфейсов подсистем.* Для каждой подсистемы определяется свой интерфейс. Только после этого возможно начать разработку самих подсистем.

На рис. 2.6 линии, соединяющие этапы, имеют стрелки на обоих концах. Это означает наличие обратной связи между этапами и возможность возвращаться к предыдущему этапу

в процессе проектирования системы. Очень часто приходится возвращаться к предыдущему этапу для решения возникших на данном этапе проблем.

Для большинства систем можно разработать несколько проектов. Это предполагает широкий диапазон возможных решений, состоящих из разных комбинаций аппаратных и программных компонентов и человеческого фактора. Для дальнейшей разработки выбирается решение, наиболее полно удовлетворяющее системным требованиям. Вместе с тем на выбор проекта часто влияют организационные и политические факторы. Например, если система разрабатывается по заказу правительства, обычно выбираются национальные поставщики комплектующих, даже если по определенным параметрам они (комплектующие) уступают зарубежным; это, естественно, влияет на выбор проекта системы.

2.4.3. Разработка подсистем

На этом этапе реализуются те подсистемы, которые были определены на этапе проектирования системы. Для отдельных подсистем этап разработки может потребовать включения различных процессов системотехники. Так, если подсистема является программной системой, этап разработки будет включать процессы формализации требований, проектирования, создания и т.д.

Иногда разработка всех подсистем начинается "с нуля". Но чаще бывает так, что некоторые подсистемы можно приобрести на рынке промышленных изделий и затем интегрировать в создаваемую систему. Обычно бывает дешевле купить готовое изделие, чем разрабатывать собственную подсистему. На этом этапе может возникнуть необходимость вернуться к этапу проектирования для того, чтобы на уровне требований "подогнать" купленное изделие к системе. Это изделие может не удовлетворять всем требованиям, предъявляемым к компоненту, который оно замещает, но если ассортимент коммерческих продуктов достаточно широк, затраты на повторное проектирование невелики.

Все подсистемы, как правило, разрабатываются параллельно. Если возникают внутренние проблемы, прерывающие процесс разработки подсистем, может потребоваться модификация всей системы. Когда система в значительной степени состоит из аппаратных компонентов, проведение модификации системы после начала производства ее компонентов может оказаться весьма затратным. Приходится находить какое-то "обходное" решение для выхода из подобных ситуаций. Часто таким решением является включение в систему программных компонентов, поскольку они достаточно гибкие и сравнительно легко поддаются модификациям. В свою очередь, это ведет к изменению требований, предъявляемых к программному обеспечению, и, как подчеркивалось в главе 1, к изменениям в проекте ПО.

2.4.4. Сборка системы

Сборка представляет собой интеграцию независимо разработанных подсистем в единую законченную систему. В процессе сборки может использоваться метод "большого взрыва", когда все подсистемы интегрируются одновременно. Но по техническим и организационным причинам гораздо предпочтительнее последовательная сборка, когда отдельные подсистемы интегрируются в систему по очереди, одна за другой.

Процесс последовательной сборки считается более подходящим для системной интеграции по двум причинам.

1. Обычно невозможно составить такой график работ, при котором у всех подсистем этап разработки заканчивается одновременно.

2. Последовательная сборка уменьшает количество ошибок, связанных с неправильной интеграцией системы. Если одновременно интегрируется несколько подсистем, то причиной обнаруженной в процессе тестирования ошибки может быть любая из них. Если интегрируется одна подсистема в уже работающую систему, то причиной обнаруженной ошибки, вероятнее всего, будет последняя интегрированная подсистема или вновь установленные связи между ней и существующими подсистемами.

Ошибки и дефекты отдельных подсистем и системы в целом часто проявляются именно на этапе сборки. Это может вызвать полемику и конфликты между разработчиками разных подсистем из-за того, чью подсистему признать "виновной" в этом. Самое неприятное в данной ситуации то, что на решение возникших проблем может потребоваться несколько недель, а то и месяцев работы.

2.4.5. Инсталляция системы

При инсталляции система "погружается" в то окружение, в котором она должна работать. В процессе инсталляции сложных систем могут проявиться различные проблемы, на решение которых подчас уходит несколько месяцев, а то и лет. Среди них могут быть проблемы, описанные ниже.

1. Окружение, в котором инсталлируется система, не совпадает с тем, для которого она спроектирована. Это общая проблема, которая часто возникает при инсталляции ПО. Например, программная система может использовать функции, которые предоставляет только определенная версия операционной системы. Однако версия, определяющая текущее окружение инсталлируемой программной системы, может не обладать этими функциями. В этом случае после инсталляции система может не работать совсем либо некоторые ее функции окажутся не реализованными.
2. Потенциальные пользователи могут относиться враждебно к внедрению этой системы в своей организации. Это может уменьшить ответственность и количество работ, необходимых для внедрения системы в данной организации. Люди могут осознанно отказываться от сотрудничества со специалистами, которые инсталлируют систему. Например, они могут отказываться от обучения работе с этой системой или не предоставлять информацию, необходимую для инсталляции системы.
3. Новая система может сосуществовать со старой до тех пор, пока в организации, где она инсталлируется, не убедятся, что новая система работает так, как требуется. Это может привести к определенным проблемам при инсталляции системы, особенно если новая и старая системы не являются полностью независимыми, а имеют некоторые общие компоненты. Случаются ситуации, когда новую систему вообще невозможно внедрить без деинсталляции старой системы. При этом испытания новой системы можно провести только тогда, когда старая система не функционирует.
4. При инсталляции возможны также чисто физические проблемы. Могут возникнуть сложности при подгонке системы к тому зданию, где она инсталлируется. Это здание может не иметь достаточного количества помещений с каналами для сетевых кабелей, могут потребоваться дополнительные воздушные кондиционеры или другие встраиваемые в здание приборы и т.п. А если это памятник истории, охраняемый законом, то при инсталляции системы изменения в здании вообще невозможны.

2.4.6. Ввод системы в эксплуатацию

После того как система инсталлирована, ее необходимо ввести в эксплуатацию. Это подразумевает проведение обучения системных операторов и изменение их обычного рабочего процесса для того, чтобы более эффективно использовать новую систему. На этом этапе могут возникнуть непредвиденные проблемы, если системная спецификация содержит ошибки или упущения. Пока система функционирует в соответствии со спецификацией, эти дефекты могут не проявиться, и поэтому разработчики могут не предусмотреть соответствующих режимов эксплуатации системы.

Например, проблемой, которая может проявиться только после ввода системы в эксплуатацию, является совместимость новой и существующих систем. Это может быть чисто физическая проблема совместимости. Возможны проблемы при передаче данных от одной системы к другой. Более "хитрой" проблемой может стать различие интерфейсов разных систем. Тогда ввод в эксплуатацию новой системы может привести к возрастанию ошибок системных операторов вследствие неправильного использования интерфейсных команд новой системы.

2.4.7. Эволюция систем

Большие и сложные системы имеют очень длительный срок жизни. В течение своей жизни они совершенствуются путем исправления ошибок в исходных системных требованиях, а также для учета новых требований, предъявляемых к ним. Вычислительные компоненты систем заменяются новыми, более производительными компонентами. Организации, эксплуатирующие систему, могут быть реорганизованы и, следовательно, использовать систему иным образом, чем предусматривалось первоначально. Может измениться внешнее окружение системы, что также требует внесения в нее изменений.

Необходимость эволюции систем, как и программного обеспечения (эта тема обсуждается в части VII), вызвана рядом причин.

1. Предполагаемые изменения в технической и деловой областях, полученные на основе тщательного анализа перспектив развития этих областей. Здесь необходимо учитывать мнение широкого круга специалистов, прежде чем принимать какие-либо решения.
2. Поскольку системы никогда не является полностью не зависимыми друг от друга, изменения в одной подсистеме обязательно окажут влияние на производительность или поведение других подсистем. Следовательно, при внесении изменений в одну подсистему необходимы изменения практически во всех подсистемах.
3. Причины, приводящие к принятию определенных решений на первоначальном этапе проектирования исходной системы, редко протоколируются или вообще фиксируются. Это может вызвать пересмотр некоторых решений, принятых на первоначальном этапе проектирования, а следовательно, изменения в самой системе.
4. По мере увеличения "возраста" системы ее структура вследствие сделанных ранее изменений нарушается, что, в свою очередь, приводит к возрастанию затрат на ее модификацию.

Вследствие все возрастающей зависимости нашего общества от систем самых разнообразных типов значительно больше усилий прилагается для совершенствования существующих систем, чем для разработки новых. Такие ранее созданные системы, которые необходимо сохранить (путем их модернизации), иногда называют *наследуемыми системами* (legasy systems). Эти системы обсуждаются в главе 26.

2.4.8. Вывод систем из эксплуатации

Вывод из эксплуатации означает извлечение системы из ее окружения после окончания срока службы. Часто это не предполагает каких-либо сложностей. Но некоторые системы могут содержать материалы, потенциально опасные для окружающей среды. Системотехники должны предусмотреть процедуру вывода такой системы из эксплуатации еще на этапе ее проектирования. Например, используемые в системе токсичные химических соединения должны быть заключены в герметические контейнеры, каждый из которых можно удалить из системы как единый элемент для последующей утилизации.

При деинсталляции программного обеспечения обычно не возникает физических проблем. Вместе с тем некоторые программные компоненты могут быть инкорпорированными в системы, необходимые для деинсталляции ПО; например, если ПО использовалось для мониторинга состояния других системных компонентов.

После вывода системы из эксплуатации некоторые ее компоненты могут использоваться в других системах. Если данные из деинсталлируемой системы должны вернуться в организацию, для этого могут быть применены какие-либо другие системы. Эти данные часто имеют значительную стоимость. Тема повторного использования данных рассматривается в главе 28.

2.5. Приобретение систем

Заказчиками сложных вычислительных систем обычно являются крупные организации, например военное ведомство, правительство или аварийные службы. Такие системы можно купить как единое целое, можно купить отдельные части, которые затем интегрируются в создаваемую систему, можно спроектировать систему и разработать по отдельному заказу "с нуля". Для больших систем процесс выбора одного из этих вариантов может растянуться на несколько месяцев или даже лет. Процесс приобретения системы — это определение наиболее оптимального для организации пути ее приобретения и выбор наилучшего поставщика системы.

Процесс приобретения системы полностью подчиняется процессу системотехники. До начала самого процесса приобретения необходимо разработать системную спецификацию и архитектуру системы, что обусловлено двумя основными причинами.

1. Для покупки или заключения контракта на разработку и построение системы необходима полностью законченная системная спецификация.
2. Практически всегда дешевле купить систему, чем разработать ее (как отдельный проект). Архитектура системы необходима для того, чтобы определить, какие ее подсистемы можно купить, а какие необходимо разрабатывать.

Большие сложные системы обычно состоят из приобретенных компонентов и компонентов, специально созданных для данной системы. Это одна из предпосылок, требующая включения программных компонентов в состав систем — программное обеспечение должно "склеить" в единое целое (причем эффективно работающее) отдельные существующие аппаратные компоненты. В необходимости разработки программного "клея" кроется причина того, что экономия от применения приобретенных компонентов не такая большая, как ожидается. Подробно тема приобретенных систем обсуждается в главе 14.

На рис. 2.7 показаны этапы процесса приобретения как готовых систем, так и разрабатываемых по заказу. Перечислим некоторые важные моменты процесса приобретения.

1. Приобретаемые компоненты, как правило, не удовлетворяют в точности всем системным требованиям, вследствие чего необходима подгонка требований в соответствии с этими компонентами. Более того, обычно стоит нелегкая дилемма выбора между системными требованиями и свойствами приобретенной системы. Чаще всего "в жертву" приносятся системные требования. Это, в свою очередь, оказывает влияние на другие подсистемы.
2. Если система разрабатывается по заказу, спецификация требований является основой контракта на приобретаемую систему. Таким образом, спецификация имеет такую же правовую силу, как и другая техническая документация.
3. После выбора разработчика системы в контракте с ним необходимо оговорить возможности внесения изменений в требования, хотя это может привести к изменению стоимости системы.

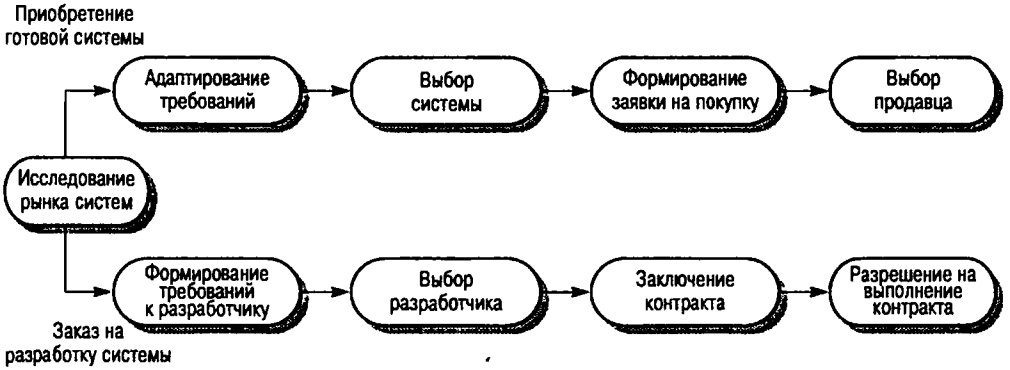


Рис. 2.7. Процесс приобретения системы

Большинство аппаратных подсистем и многие программные подсистемы (такие, как системы управления базами данных) не разрабатываются специально для включения в состав больших систем. Часто в них встраиваются уже готовые системы.

Очень немногие организации имеют возможности для проектирования, производства и тестирования всех компонентов сложных больших систем. Организация – разработчик системы, которую обычно называют ведущим или генеральным подрядчиком, может заключать контракты на разработку отдельных подсистем с другими субподрядчиками (рис. 2.8). Для создания больших систем, таких как системы управления полетами, группа организаций-разработчиков может создать консорциум для выполнения контракта. В консорциум должны входить разработчики и поставщики всех компонентов системы, например разработчики вычислительных устройств и программного обеспечения, поставщики периферийного оборудования и специального оборудования (скажем, радаров).

Модель "подрядчик-субподрядчик" минимизирует количество организаций, участвующих в реализации контракта. Субподрядчики разрабатывают и производят части системы в соответствии со спецификацией, предоставляемой ведущим подрядчиком. После завершения работ субподрядчиками система собирается из отдельных частей ведущим подрядчиком. Готовая система поставляется заказчику (покупателю). В зависимости от условий контракта, заказчик может предоставить ведущему подрядчику свободный выбор субподрядчиков либо потребовать, чтобы субподрядчики выбирались из заранее оговоренного списка.

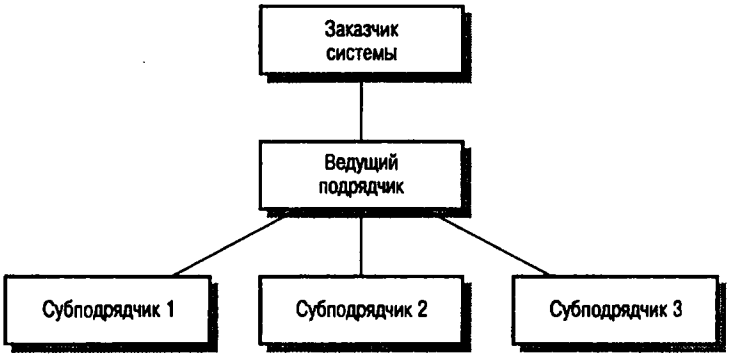


Рис. 2.8. Модель "подрядчик–субподрядчик"

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Системотехника — это комплексная технология создания систем, которая требует привлечения многих инженерных дисциплин.
- Интегрированные свойства системы — это свойства, которые присущи системе как единому целому, а не ее отдельным компонентам. К интегрированным системным свойствам относятся безотказность, производительность, удобство эксплуатации, безопасность и защищенность системы.
- Архитектура системы, обычно представляемая в виде блок-схемы, показывает основные подсистемы и их взаимосвязь.
- Функциональные компоненты систем делятся на следующие типы: сенсорные, исполнительные, вычислительные, координирующие, коммуникационные и интерфейсные.
- Процесс создания систем включает следующие этапы: составление спецификации, проектирование, разработку, интеграцию (сборку) и тестирование. Наиболее ответственным этапом является сборка системы, когда различные подсистемы, подчас от разных производителей, интегрируются в единую систему.
- Процесс приобретения системы включает этапы специфицирования системы, формирование заявки на приобретение, выбор поставщика и затем заключение контракта на покупку или разработку системы. Часто некоторые части вычислительных систем приобретаются у сторонних производителей.

Упражнения

2.1. Объясните, почему системное окружение может оказать непредвиденное воздействие на функционирование системы.

2.2. Измените схему на рис. 2.6 таким образом, чтобы включить в нее этап приобретения подсистем после этапа их идентификации. Покажите на новой схеме обратную связь от включенного этапа приобретения к другим этапам процесса проектирования системы.

2.3. Объясните, почему процесс специфицирования систем, используемых аварийными службами для управления в чрезвычайных ситуациях, является "злостной" проблемой.

2.4. Объясните важность получения полного описания системной архитектуры на самой ранней стадии процесса разработки системной спецификации.

2.5. На рис. 2.1 показана иерархия систем отдельного здания. Система безопасности, включающая систему защиты от несанкционированного проникновения в здание и противопожарную систему, является расширением системы, представленной на рис. 2.2. Она содержит датчики дыма, датчики дви-

жения и дверные датчики, видеокамеры, управляемые компьютером, расположенные в разных местах строения, пульт управления, где собирается вся информация от системы безопасности, средства внешних коммуникаций для связи с соответствующими службами, такими как полиция и пожарная охрана. Нарисуйте блок-схему архитектуры такой системы.

2.6. Разрабатывается система предупреждения наводнений для города, которому угрожают частые наводнения. Система включает множество датчиков уровня воды в реке, связь с метеослужбой, предоставляющей прогноз погоды, связь со службами безопасности (полицией, береговой охраной и т.д.), видеомониторы, установленные в различных местах, и комнату управления, оборудованную пультом управления и мониторами.

2.7. Дежурные операторы имеют доступ к базе данных и могут переключать видеомониторы. База данных содержит информацию с датчиков, расположенных в других городах, также подверженных риску наводнения, о ситуации в этих городах (уровень воды, сила и направление ветра и т.п.), таблицу высот прибрежных городов, местоположение оборудования, контролирующего уровень воды, контактные телефоны служб безопасности, частоты местных радиостанций и т.д.

2.8. Нарисуйте блок-схему возможной архитектуры такой системы. Также определите основные подсистемы и взаимосвязи между ними.

2.9. Назовите три проблемы, которые могут возникнуть при инсталляции системы в сторонней организации.

2.10. Рассмотрите системотехнику как профессию и сравните ее с профессией инженера-электронщика и специалиста по программному обеспечению.

2.11. Допустим, вы инженер, включенный в группу разработчиков финансовой системы. В процессе инсталляции системы вы обнаруживаете, что ее внедрение в организации может привести к увольнению большого числа людей. Персонал организации не предоставляет вам информацию, необходимую для завершения инсталляции системы. Что вы будете делать в этой ситуации как инженер-системотехник? Будете ли вы с чувством профессиональной ответственности стремиться к завершению ее внедрения системы, что требуется от вас контрактом? Или же просто прекратите работу до тех пор, пока организация не разберется со своими проблемами?

3

Процесс создания программного обеспечения

Цели

Цель настоящей главы — представить основные идеи лежащие в основе процесса создания программного обеспечения. Прочитав эту главу, вы должны:

- знать основные концепции, лежащие в основе процесса создания ПО и моделей этого процесса;
- иметь представление об основных моделях процесса создания ПО и понимать, когда какую из них использовать;
- знать схему построения моделей процесса формирования требований к ПО, его разработки тестирования и модернизации;
- иметь понятие о CASE-технологиях, предназначенных для поддержки процесса создания ПО.

Содержание

Как отмечалось в главе 1, процесс создания программного обеспечения — это множество взаимосвязанных процессов и результатов их выполнения, которые ведут к созданию программного продукта. Процесс создания ПО может начинаться с разработки программной системы "с нуля", но чаще новое ПО разрабатывается на основе существующих программных систем путем их модификации.

Процесс создания ПО, как и любая другая интеллектуальная деятельность, основан на человеческих суждениях и умозаключениях, т.е. является творческим. Вследствие этого все попытки автоматизировать процесс создания ПО имеют лишь ограниченный успех. CASE-средства (обсуждаются в разделе 3.7) могут помочь в реализации некоторых этапов процесса разработки ПО, но по крайней мере в ближайшие несколько лет не стоит ожидать от них существенного продвижения в автоматизации тех этапов создания ПО, где существенен фактор творческого подхода к разработке ПО.

Одна из причин ограниченного применения автоматизированных средств к процессу создания ПО — огромное многообразие видов деятельности, связанных с разработкой программных продуктов. Кроме того, организации-разработчики используют разные подходы к разработке ПО. Также различаются характеристики и возможности создаваемых систем, что требует особого внимания к определенным сторонам процесса разработки. Поэтому даже в одной организации при создании разных программных систем могут использоваться различные подходы и технологии.

Несмотря на то что наблюдается огромное многообразие подходов, методов и технологий создания ПО, существуют фундаментальные базовые процессы, без реализации которых не может обойтись ни одна технология разработки программных продуктов. Перечислим эти процессы.

1. *Разработка спецификации ПО.* Это фундамент любой программной системы. Спецификация определяет все функции и действия, которые будет выполнять разрабатываемая система.
2. *Проектирование и реализация (производство) ПО.* Это процесс непосредственного создания ПО на основе спецификации.
3. *Аттестация ПО.* Разработанное программное обеспечение должно быть аттестовано на соответствие требованиям заказчика.
4. *Эволюция ПО.* Любые программные системы должны модифицироваться в соответствии с изменениями требований заказчика.

В данной главе приведен обзор этих процессов, более подробно они рассматриваются в последующих частях книги.

Хотя не существует "идеального" процесса создания ПО, во многих организациях-разработчиках пытаются его усовершенствовать, поскольку он может опираться на устаревшие технологии и не включать лучших методов современной инженерии программного обеспечения. Кроме того, многие организации постоянно используют одни и те же технологии (когда-то ранее хорошо себя зарекомендовавшие) и им также необходимы методы современной инженерии ПО.

Совершенствовать процесс создания программных систем можно разными путями. Например, путем стандартизации, которая уменьшит разнородность используемых в данной организации технологий. Это, в свою очередь, приведет к совершенствованию внутренних коммуникаций в организации, уменьшению времени обучения персонала и сделает экономически выгодным процесс автоматизации разработок. Стандартизация обычно является первым шагом к внедрению новых методов и технологий инженерии ПО. Тема совершенствования процесса разработки программных продуктов освещается в главе 25.

3.1. Модели процесса создания ПО

Как отмечалось в главе 1, модель процесса создания программного обеспечения – это общее абстрактное представление данного процесса. Каждая такая модель представляет процесс создания ПО в каком-то своем "разрезе", используя только определенную часть всей информации о процессе. В настоящем разделе представлены обобщенные модели, основанные на архитектурном подходе. В этом случае можно увидеть всю структуру процесса создания ПО, абстрагируясь от частных деталей отдельных составляющих его этапов.

Эти обобщенные модели не содержат точного описания всех стадий процесса создания ПО. Напротив, они являются полезными абстракциями, помогающими "приложить" различные подходы и технологии к процессу разработки. Кроме того, очевидно, что процесс создания больших систем не является единым, а состоит из множества различных процессов, ведущих к созданию отдельных частей большой системы.

В этой главе рассматриваются следующие модели создания программного обеспечения.

1. *Каскадная модель.* Основные базовые виды деятельности, выполняемые в процессе создания ПО (такие, как разработка спецификации, проектирование и производство, аттестация и модернизация ПО), представляются как отдельные этапы этого процесса.
2. *Эволюционная модель разработки ПО.* Здесь последовательно перемежаются этапы формирования требований, разработки ПО и его аттестации. Первоначальная программная система быстро разрабатывается на основе некоторых абстрактных общих требований. Затем они уточняются и детализируются в соответствии с требованиями заказчика. Далее система дорабатывается и аттестуется в соответствии с новыми уточненными требованиями.
3. *Модель формальной разработки систем.* Основана на разработке формальной математической спецификации программной системы и преобразовании этой спецификации посредством специальных математических методов в исполняемые программы. Проверка соответствия спецификации и системных компонентов также выполняется математическими методами.
4. *Модель разработки ПО на основе ранее созданных компонентов.* Предполагает, что отдельные составные части программной системы уже существуют, т.е. созданы ранее. В этом случае технологический процесс создания ПО основное внимание уделяет интеграции отдельных компонентов в общее целое, а не их созданию.

Каскадная и эволюционная модели разработки широко используются на практике. Модель формальной разработки систем успешно применялась во многих проектах [219, 239, 8*, 18*], но количество организаций-разработчиков, постоянно использующих этот метод, невелико. Использование готовых системных компонентов практикуется повсеместно, но большинство организаций не придерживаются в точности модели разработки ПО на основе ранее созданных компонентов. Вместе с тем этот метод должен получить широкое распространение в XXI столетии, поскольку сборка систем из готовых или ранее использованных компонентов значительно ускоряет разработку ПО. Эта модель рассматривается в главе 14.

3.1.1. Каскадная модель

Это первая модель процесса создания ПО, порожденная моделями других инженерных процессов [300]. Она показана на рис. 3.1. Эту модель также иногда называют моделью жизненного цикла программного обеспечения[1] . Основные принципиальные этапы (стадии) этой модели отражают все базовые виды деятельности, необходимые для создания ПО.

---

[1] *Жизненный цикл программного обеспечения – это совокупность процессов, протекающих в период от момента принятия решения о создании ПО до его полного вывода из эксплуатации. Таким образом, "жизненный*

1. *Анализ и формирование требований.* Путем консультаций с заказчиком ПО определяются функциональные возможности, ограничения и цели создаваемой программной системы.
2. *Проектирование системы и программного обеспечения.* Процесс проектирования системы разбивает системные требования на требования, предъявляемые к аппаратным средствам, и требования к программному обеспечению системы. Разрабатывается общая архитектура системы. Проектирование ПО предполагает определение и описание основных программных компонентов и их взаимосвязей.
3. *Кодирование и тестирование программных модулей.* На этой стадии архитектура ПО реализуется в виде множества программ или программных модулей. Тестирование каждого модуля включает проверку его соответствия требованиям к данному модулю.
4. *Сборка и тестирование системы.* Отдельные программы и программные модули интегрируются и тестируются в виде целостной системы. Проверяется, соответствует ли система своей спецификации.
5. *Эксплуатация и сопровождение системы.* Обычно (хотя и не всегда) это самая длительная фаза жизненного цикла ПО. Система инсталлируется, и начинается период ее эксплуатации. Сопровождение системы включает исправление ошибок, которые не были обнаружены на более ранних этапах жизненного цикла, совершенствование системных компонентов и "подгонку" функциональных возможностей системы к новым требованиям.

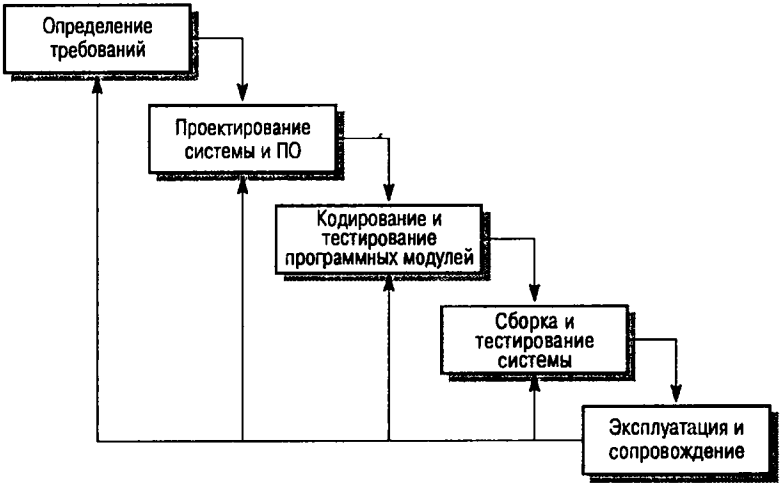


Рис. 3.1. Жизненный цикл программного обеспечения

В принципе результат каждого этапа должен утверждаться документально (это как бы сигнал об окончании этапа). Тогда следующий этап не может начаться до завершения предыдущего. Однако на практике этапы могут перекрываться с постоянным перетеканием информации от одного этапа к другому. Например, на этапе проектирования может возникнуть необходимость уточнить системные требования либо на этапе кодирования могут выявиться проблемы, которые можно решить лишь на этапе проектирования, и т.д. Процесс создания

цикл ПО" является более широким понятием, чем модель процесса создания ПО. Вместе с тем каскадную модель можно рассматривать как одну из моделей жизненного цикла ПО. – Прим. ред.

ПО нельзя описать простой линейной моделью, так как он неизбежно содержит последовательность повторяющихся процессов.

Поскольку на каждом этапе проводятся определенные работы и оформляется сопутствующая документация, повторение этапов приводит к повторным работам и значительным расходам. Поэтому после небольшого числа повторений обычно "замораживается" часть этапов создания ПО, например этап определения требований, но продолжается выполнение последующих этапов. Возникающие проблемы, решение которых требует возврата к "замороженным" этапам, игнорируются либо делаются попытки решить их программно. "Замораживание" этапа определения требований может привести к тому, что разработанная система не будет удовлетворять всем требованиям заказчика. Это также может привести к появлению плохо структурированной системы, если упущения этапа проектирования исправляются только с помощью программистских хитростей.

Последний этап жизненного цикла ПО (эксплуатация и сопровождение) – это "полноценное" использование программной системы. На этом этапе могут обнаружиться ошибки, допущенные, например, на первом этапе формирования требований. Могут также проявиться ошибки проектирования и кодирования, что может потребовать определения новых функциональных возможностей системы. С другой стороны, система постоянно должна оставаться работоспособной. Внесение необходимых изменений в программную систему может потребовать повторения некоторых или даже всех этапов процесса создания ПО.

К недостаткам каскадной модели можно отнести негибкое разбиение процесса создания ПО на отдельные фиксированные этапы. В этой модели определяющие систему решения принимаются на ранних этапах и затем их трудно отменить или изменить, особенно это относится к формированию системных требований. Поэтому каскадная модель применяется тогда, когда требования формализованы достаточно четко и корректно. Вместе с тем каскадная модель хорошо отражает практику создания ПО. Технологии создания ПО, основанные на данной модели, используются повсеместно, в частности для разработки систем, входящих в состав больших инженерных проектов.

3.1.2. Эволюционная модель разработки

Эта модель основана на следующей идее: разрабатывается первоначальная версия программного продукта, которая передается на испытание пользователям, затем она дорабатывается с учетом мнения пользователей, получается промежуточная версия продукта, которая также проходит "испытание пользователем", снова дорабатывается и так несколько раз, пока не будет получен необходимый программный продукт (рис. 3.2). Отличительной чертой данной модели является то, что процессы специфицирования, разработки и аттестации ПО выполняются параллельно при постоянном обмене информацией между ними.

Различают два подхода к реализации эволюционного метода разработки.

1. *Подход пробных разработок.* Здесь большую роль играет постоянная работа с заказчиком (или пользователями) для того, чтобы определить полную систему требований к ПО, необходимую для разработки конечной версии продукта. В рамках этого подхода вначале разрабатываются те части системы, которые очевидны или хорошо специфицированы. Система эволюционирует (дорабатывается) путем добавления новых средств по мере их предложения заказчиком.
2. *Прототипирование.* Здесь целью процесса эволюционной разработки ПО является поэтапное уточнение требований заказчика и, следовательно, получение законченной спецификации, определяющей разрабатываемую систему. Прототип[2] обычно строится для экспериментирования с той частью требований заказчика, которые сформированы нечетко или с внутренними противоречиями.

---

[2] *Под прототипом обычно понимается действующий программный модуль, реализующий отдельные функции создаваемого ПО.* – Прим. ред.

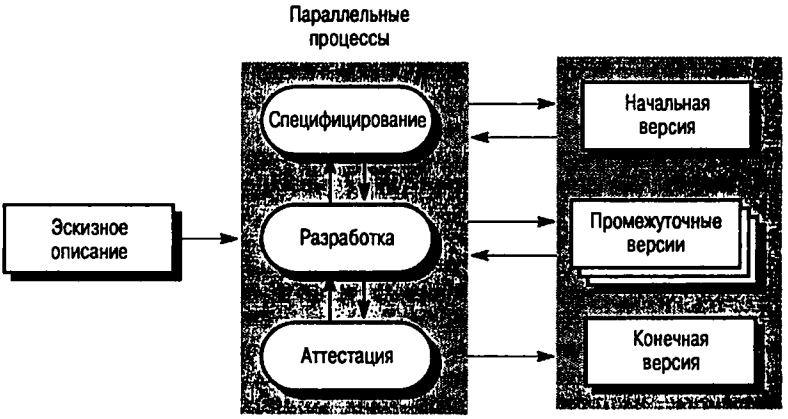


Рис. 3.2. Эволюционная модель разработки

Более подробно подходы к эволюционной разработке рассматриваются в главе 8, где описаны различные технологии прототипирования систем.

Эволюционный подход часто более эффективен, чем подход, построенный на основе каскадной модели, особенно если требования заказчика могут меняться в процессе разработки системы. Достоинством процесса создания ПО, построенного на основе эволюционного подхода, является то, что здесь спецификация может разрабатываться постепенно, по мере того как заказчик (или пользователи) осознает и сформулирует те задачи, которые должно решать программное обеспечение. Вместе с тем данный подход имеет и некоторые недостатки.

1. *Многие этапы процесса создания ПО не документированы.* Менеджерам проекта создания ПО необходимо регулярно документально отслеживать выполнение работ. Но если система разрабатывается быстро, то экономически не выгодно документировать каждую версию системы.
2. *Система часто получается плохо структурированной.* Постоянные изменения в требованиях приводят к ошибкам и упущениям в структуре ПО. Со временем внесение изменений в систему становится все более сложным и затратным.
3. *Часто требуются специальные средства и технологии разработки ПО.* Это вызвано необходимостью быстрой разработки версий программного продукта. Но, с другой стороны, это может привести к несовместимости некоторых применяемых средств и технологий, что, в свою очередь, требует наличия в команде разработчиков специалистов высокого уровня.

Я думаю, что эволюционный подход наиболее приемлем для разработки небольших программных систем (до 100 000 строк кода) и систем среднего размера (до 500 000 строк кода) с относительно коротким сроком жизни. На больших долгоживущих системах слишком заметно проявляются недостатки этого подхода. Для таких систем я рекомендую смешанный подход к созданию ПО, который вобрал бы в себя лучшие черты каскадной и эволюционной моделей разработки.

При таком смешанном подходе для прояснения "темных мест" в системной спецификации можно использовать прототипирование. Часть системных компонентов, для которых полностью определены требования, может создаваться на основе каскадной модели. Другие системные компоненты, которые трудно поддаются специфицированию, например пользовательский интерфейс, могут разрабатываться с использованием прототипирования.

3.1.3. Формальная разработка систем

Этот подход к созданию ПО имеет много черт, сходных с каскадной моделью, но построен на основе формальных математических преобразований системной спецификации в исполняемую программу. Процесс создания программного обеспечения в соответствии с этим подходом показан на рис. 3.3. Здесь для упрощения не указаны обратные связи между этапами процесса.

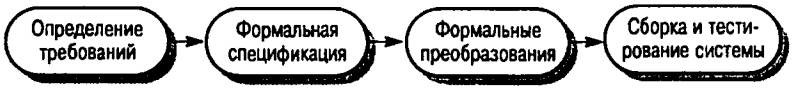


Рис. 3.3. Модель формальной разработки ПО

Между данным подходом и каскадной моделью существуют следующие кардинальные отличия.

1. Здесь спецификация системных требований имеет вид детализированной формальной спецификации, записанной с помощью специальной математической нотации.
2. Процессы проектирования, написания программного кода и тестирования системных модулей заменяются процессом, в котором формальная спецификация путем последовательных формальных преобразований трансформируется в исполняемую программу. Этот процесс показан на рис. 3.4.

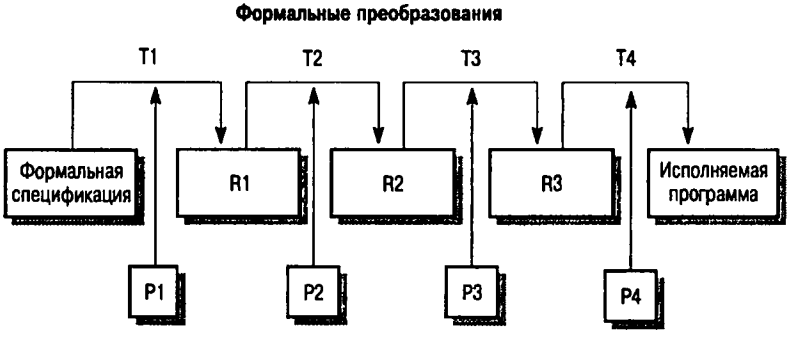


Рис. 3.4. Процесс формальных преобразований

В процессе преобразования формальное математическое представление системы последовательно и математически корректно трансформируется в программный код, постепенно все более детализированный. Эти преобразования выполняются до тех пор, пока все позиции формальной спецификации не будут трансформированы в эквивалентную программу. Преобразования выполняются математически корректно – здесь не существует проблемы проверки соответствия спецификации и программы.

Преимущество метода формальных преобразований, которое заключается в точном соответствии конечной программы спецификации, обеспечивается тем, что дистанция между последовательными преобразованиями значительно меньше дистанции между спецификацией и программой. Доказательство корректности программного кода для больших масштабируемых систем обычно очень длительно, а часто просто не выполнимо. В этом отношении метод формальных преобразований, состоящий из последовательности небольших формальных шагов, весьма привлекателен. Однако выбор для применения соответствующих формальных преобразований сложен и неочевиден.

Наиболее известным примером метода формальных преобразований является метод "чистой комнаты" (Cleanroom), разработанный компанией IBM [239, 310, 219, 284]. Этот метод предполагает пошаговую разработку ПО, когда на каждом шаге применяется формальные преобразования. Это позволяет отказаться от тестирования отдельных программных модулей, а тестирование всей системы происходит после ее сборки. Этот подход более подробно рассмотрен в главе 19.

Как метод "чистой комнаты", так и другие методы формальных преобразований основываются на B-методе [348]. Все эти методы имеют несколько "врожденных" недостатков, а стоимость разработки ПО с помощью этих методов не намного отличается от стоимости разработок другими методами. Методы формальных преобразований обычно применяют для разработки систем, которые должны удовлетворять строгим требованиям надежности, безотказности и безопасности, так как они гарантируют соответствие созданных систем их спецификациям.

Кроме разработки указанного типа систем, методы формальных преобразований не нашли широкого применения, поскольку требуют специальных знаний и опыта использования. Кроме того, для большинства систем эти методы не дают существенного выигрыша в стоимости или качестве по сравнению с другими методами разработки ПО. Основная причина заключается в том, что функционирование большинства систем с трудом поддается описанию методом формальных спецификаций, – при создании большинства программных систем большая часть усилий разработчиков уходит именно на создание спецификаций.

3.1.4. Разработка ПО на основе ранее созданных компонентов

В большинстве программных проектов применяется повторное использование некоторых программных модулей. Это обычно случается там, где разработчики проекта знают о ранее созданных программных продуктах, в составе которых есть компоненты, приблизительно удовлетворяющие требованиям разрабатываемых компонентов. Эти компоненты модифицируются в соответствии с новыми требованиями и затем включается в состав новой системы. В эволюционной модели разработки, описанной в разделе 3.1.2, для ускорения процесса создания ПО повторное использование ранее созданных компонентов применяется достаточно часто.

Неформальное решение о повторном использовании ранее созданных программных компонентов обычно принимается независимо от общего процесса создания ПО. Вместе с тем на протяжении нескольких последних лет все более широко применяется подход к созданию ПО, основанный именно на повторном использовании ранее созданных программных модулей.

Этот подход основан на наличии большой базы существующих программных компонентов, которые можно интегрировать в создаваемую новую систему. Часто такими компонентами являются свободно продаваемые на рынке программные продукты, которые можно использовать для выполнения определенных специальных функций, таких как форматирование текста, числовые вычисления и т.п. Общая модель процесса разработки ПО с повторным использованием ранее созданных компонентов показана на рис. 3.5.

В этом подходе начальный этап специфицирования требований и этап аттестации такие же, как и в других моделях процесса создания ПО. А этапы, расположенные между ними, имеют следующий смысл.

1. *Анализ компонентов.* Имея спецификацию требований, на этом этапе осуществляется поиск компонентов, которые могли бы удовлетворить сформулированным требова-

ниям. Обычно невозможно точно сопоставить функции, реализуемые готовыми компонентами, и функции, определенные спецификацией требований.

2. *Модификация требований.* На этой стадии анализируются требования с учетом информации о компонентах, полученной на предыдущем этапе. Требования модифицируются таким образом, чтобы максимально использовать возможности отобранных компонентов. Если изменение требований невозможно, повторно выполняется анализ компонентов для того, чтобы найти какое-либо альтернативное решение.
3. *Проектирование системы.* На данном этапе проектируется структура системы либо модифицируется существующая структура повторно используемой системы. Проектирование должно учитывать отобранные программные компоненты и строить структуру в соответствии с их функциональными возможностями. Если некоторые готовые программные компоненты недоступны, проектируется новое ПО.
4. *Разработка и сборка системы.* Это этап непосредственного создания системы. В рамках рассматриваемого подхода сборка системы является скорее частью разработки системы, чем отдельным этапом.

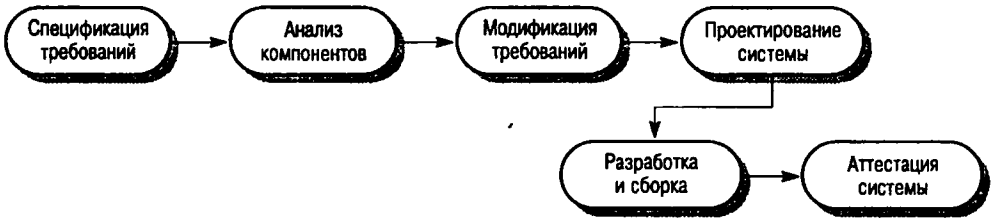


Рис. 3.5. Разработка ПО с повторным использованием ранее созданных компонентов

Основные достоинства описываемой модели процесса разработки ПО с повторным использованием ранее созданных компонентов заключаются в том, что сокращается количество непосредственно разрабатываемых компонентов и уменьшается общая стоимость создаваемой системы. Вместе с тем при использовании этого подхода неизбежны компромиссы при определении требований; это может привести к тому, что законченная система не будет удовлетворять всем требованиям заказчика. Кроме того, при проведении модернизации системы (т.е. при создании ее новой версии) отсутствует возможность влиять на появление новых версий компонентов, используемых в системе, что значительно затрудняет сам процесс модернизации.

3.2. Итерационные модели разработки ПО

Описанные модели процесса создания ПО имеют свои достоинства и недостатки. При создании больших систем, как правило, приходится использовать различные подходы к разработке разных частей системы, т.е. в целом к разработке системы применяются гибридные (смешанные) модели. Поэтому важную роль играет возможность выполнять отдельные процессы разработки подсистем и весь процесс создания ПО итерационно, когда в ответ на изменения требований повторно выполняются определенные этапы создания системы (чаще всего этапы проектирования и кодирования).

В этом разделе я представлю две гибридные итерационные модели, сочетающие несколько различных подходов к разработке ПО и разработанные специально для поддержки итерационного способа создания ПО.

1. *Модель пошаговой разработки*, где процессы специфицирования требований, проектирования и написания кода разбиваются на последовательность небольших шагов, которые ведут к созданию ПО.
2. *Спиральная модель разработки*, в которой весь процесс создания ПО, от начального эскиза системы до ее конечной реализации, разворачивается по спирали.

Существенным отличием итерационных моделей является то, что здесь процесс разработки спецификации протекает параллельно с разработкой самой программной системы. Более того, в модели пошаговой разработки полную системную спецификацию можно получить только после завершения самого последнего шага процесса создания ПО. Очевидно, что такой подход входит в противоречие с моделью приобретения ПО, когда полная системная спецификация является составной частью контракта на разработку системы. Поэтому, чтобы применять итерационные модели для разработки больших систем, которые заказываются "серьезными" организациями, например государственными агентствами, необходимо менять форму контракта, на что такие организации идут с большим трудом.

3.2.1. Модель пошаговой разработки

В каскадной модели создания ПО определение требований осуществляется совместно с заказчиком до начала проектирования системы, точно так же системная архитектура должна быть создана до начала непосредственной реализации (кодирования) системы. Изменения в требованиях, сделанные на этапе написания кода, ведут к необходимости выполнения повторных работ по проектированию и кодированию системы. Вместе с тем к достоинствам каскадной модели можно отнести простоту управления процессом создания ПО (в рамках данной модели), а также наличие отдельных этапов проектирования и реализации, что приводит к созданию вполне работоспособных систем, в которых учтены все изменения в спецификации, сделанные уже во время самого процесса разработки ПО. В отличие от каскадной, в эволюционной модели можно отложить принятие окончательных решений о спецификации и структуре системы, однако это может привести к созданию плохо структурированной системы, которая будет также трудна в сопровождении.

Модель пошаговой разработки находится где-то между этими моделями и объединяет их достоинства. Эта модель (рис. 3.6) была предложена Миллсом (Mills, [240]) как попытка уменьшить количество повторно выполняемых работ в процессе создания ПО и увеличить для заказчика временной период окончательного принятия решения обо всех деталях системных требований.

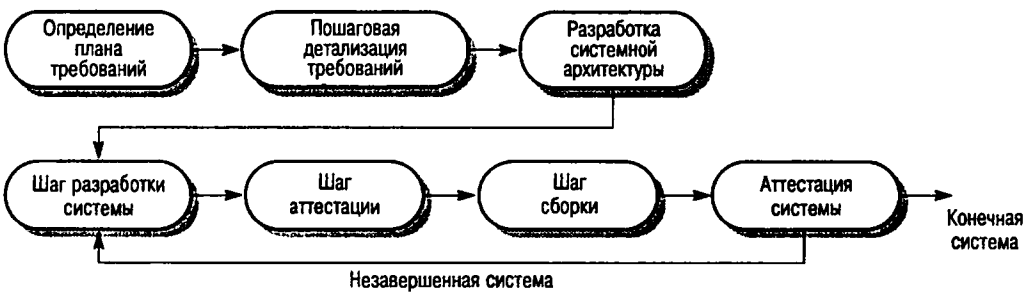


Рис. 3.6. Модель пошаговой разработки

В процессе пошаговой разработки заказчик сначала в общих чертах определяет те сервисы (функциональные возможности), которые должны присутствовать у создаваемой системы. При этом устанавливаются приоритеты, т.е. определяется, какие сервисы более важны, а какие — менее. Также определяется количество шагов разработки, причем на каждом шаге должен быть получен системный компонент, реализующий определенное подмножество системных функций. Распределение реализации системных сервисов по шагам разработки зависит от их приоритетов. Сервисы с более высокими приоритетами реализуются первыми.

Последовательность шагов разработки определяется заранее до начала их выполнения. На первых шагах детализируются требования для сервисов, затем для их реализации (на последующих шагах) используется один из подходящих способов разработки ПО. В ходе их реализации анализируются и детализируются требования для компонентов, которые будут разрабатываться на более поздних шагах, причем изменение требований для тех компонентов, которые уже находятся в процессе разработки, не допускается.

После завершения шага разработки получаем программный компонент, который передается заказчику для интегрирования в подсистему, реализующую определенный системный сервис. Заказчик может экспериментировать с готовыми подсистемами и компонентами для того, чтобы уточнить требования, предъявляемые к следующим версиям уже готовых компонентов или к компонентам, разрабатываемым на последующих шагах. По завершении очередного шага разработки полученный компонент интегрируется с ранее произведенными компонентами; таким образом, после каждого шага разработки система приобретает все большую функциональную завершенность. Общесистемные функции в этом процессе могут реализоваться сразу или постепенно, по мере разработки необходимых компонентов.

В описываемой модели не предполагается, что на каждом шаге используется один и тот же подход к процессу разработки компонентов. Если создаваемый компонент имеет хорошо разработанную спецификацию, то для его создания можно применить каскадную модель. Если же требования определены нечетко, можно использовать эволюционную модель разработки.

Процесс пошаговой разработки имеет целый ряд достоинств.

1. Заказчику нет необходимости ждать полного завершения разработки системы, чтобы получить о ней представление. Компоненты, полученные на первых шагах разработки, удовлетворяют наиболее критическим требованиям (так как имеют наибольший приоритет) и их можно оценить на самой ранней стадии создания системы.
2. Заказчик может использовать компоненты, полученные на первых шагах разработки, как прототипы и провести с ними эксперименты для уточнения требований к тем компонентам, которые будут разрабатываться позднее.
3. Данный подход уменьшает риск общесистемных ошибок. Хотя в разработке отдельных компонентов возможны ошибки, но эти компоненты должны пройти соответствующее тестирование и аттестацию, прежде чем их передадут заказчику.
4. Поскольку системные сервисы с высоким приоритетом разрабатываются первыми, а все последующие компоненты интегрируются с ними, неизбежно получается так, что наиболее важные подсистемы подвергаются более тщательному всестороннему тестированию и проверке. Это значительно снижает вероятность программных ошибок в особо важных частях системы.

Вместе с тем при реализации пошаговой разработки могут возникнуть определенные проблемы. Компоненты, получаемые на каждом шаге разработки, имеют относительно небольшой размер (обычно не более 20 000 строк кода), но должны реализовать какую-либо системную функцию. Отобразить множество системных требований к компонентам нужного размера довольно сложно. Более того, многие системы должны обладать набором базовых системных свойств, которые реализуются совместно различными частями системы. Поскольку требования детально не определены до тех пор, пока не будут разработаны все компоненты, бывает весьма сложно распределить общесистемные функции по компонентам.

В настоящее время предложен метод так называемого экстремального программирования (extreme programming), который устраняет некоторые недостатки метода пошаговой разработки. Этот метод основан на пошаговой разработке малых программных компонентов, реализующих небольшие функциональные требования, постоянном вовлечении заказчика в процесс разработки и обезличенном программировании (см. главу 23). В статье [31] описан метод экстремального программирования и приведено несколько отчетов о его успешном применении, однако подобные отчеты можно привести для всех основных методов разработки ПО.

3.2.2. Спиральная модель разработки

Спиральная модель процесса создания программного обеспечения (рис. 3.7) была впервые предложена в статье [47] и в настоящее время получила широкую известность и популярность. В отличие от рассмотренных ранее моделей, где процесс создания ПО представлен в виде последовательности отдельных процессов с возможной обратной связью между ними, здесь процесс разработки представлен в виде спирали. Каждый виток спирали соответствует одной стадии (итерации) процесса создания ПО. Так, самый внутренний виток спирали соответствует стадии принятия решения о создании ПО, на следующем витке определяются системные требования, далее следует стадия (виток спирали) проектирования системы и т.д.

Каждый виток спирали разбит на четыре сектора.

1. *Определение целей.* Определяются цели каждой итерации проекта. Кроме того, устанавливаются ограничения на процесс создания ПО и на сам программный продукт, уточняются планы производства компонентов. Определяются проектные риски (например, риск превышения сроков или риск превышения стоимости проекта). В зависимости от "проявленных" рисков, могут планироваться альтернативные стратегии разработки ПО.
2. *Оценка и разрешение рисков.* Для каждого определенного проектного риска проводится его детальный анализ. Планируются мероприятия для уменьшения (разрешения) рисков. Например, если существует риск, что системные требования определены неверно, планируется разработать прототип системы.
3. *Разработка и тестирование.* После оценки рисков выбирается модель процесса создания системы. Например, если доминируют риски, связанные с разработкой интерфейсов, наиболее подходящей будет эволюционная модель разработки ПО с прототипированием. Если основные риски связаны с соответствием системы и спецификации, скорее всего, следует применить модель формальных преобразований. Каскадная модель может быть применена в том случае, если основные риски определены как ошибки, которые могут проявиться на этапе сборки системы.

4. *Планирование.* Здесь пересматривается проект и принимается решение о том, начинать ли следующий виток спирали. Если принимается решение о продолжении проекта, разрабатывается план на следующую стадию проекта.

Существенное отличие спиральной модели от других моделей процесса создания ПО заключается в точном определении и оценивании рисков. Если говорить неформально, то риск — это те неприятности, которые могут случиться в процессе разработки системы. Например, если при написании программного кода используется новый язык программирования, то риск может заключаться в том, что компилятор этого языка может быть ненадежным или что результирующий код может быть не достаточно эффективным. Риски могут также заключаться в превышении сроков или стоимости проекта. Таким образом, уменьшение (разрешение) рисков — важный элемент управления системным проектом. В главе 4, посвященной управлению программными проектами, возможные риски и способы их разрешения рассматриваются более детально.

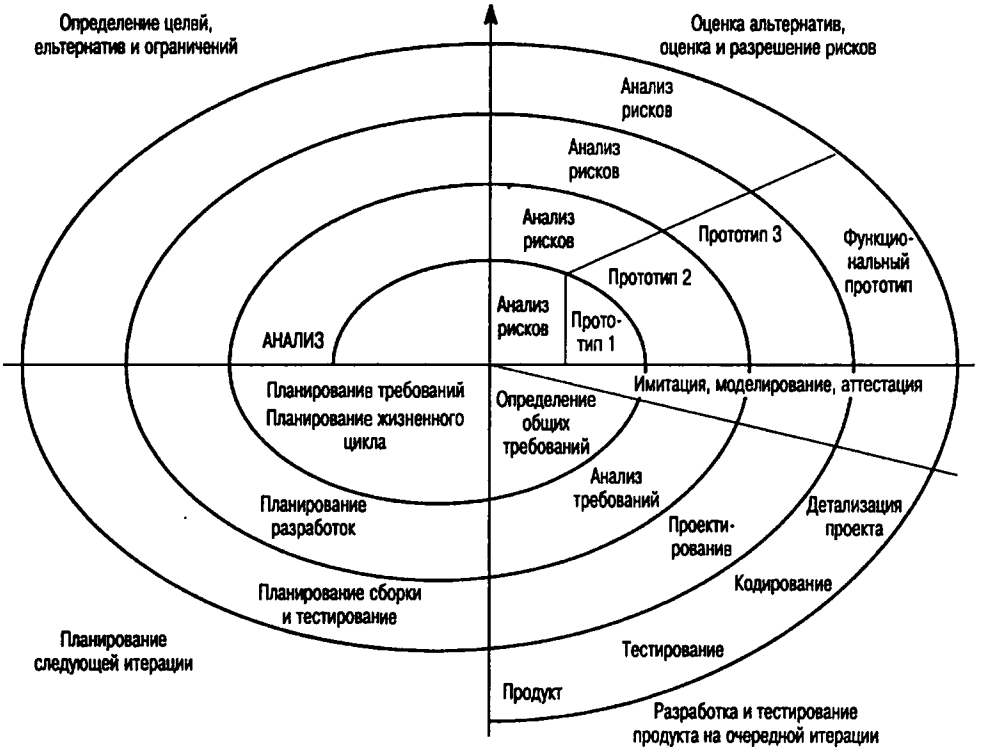


Рис. 3.7. Спиральная модель создания ПО (© 1988 IEEE)

Первая итерация создания ПО в спиральной модели начинается с тщательной проработки системных показателей (целей системы), таких как эксплуатационные показатели и функциональные возможности системы. Конечно, альтернативных путей достижения этих показателей или целей можно сформировать бесконечно много. Но каждая альтернатива должна оценивать стоимость достижения каждой сформулированной цели. Результаты анализа возможных альтернатив служат источником оценки проектного риска. Это происходит на следующей стадии спиральной модели, где для оценки рисков

используются более детальный анализ альтернатив, прототипирование, имитационное моделирование и т.п. С учетом полученных оценок рисков выбирается тот или иной подход к разработке системных компонентов, далее он реализуется, затем осуществляется планирование следующего этапа процесса создания ПО.

В спиральной модели нет фиксированных этапов, таких как разработка спецификации или проектирование. Эта модель может включать в себя любые другие модели разработки систем. Например, на одном витке спирали может использоваться протипирование для более четкого определения требований (и, следовательно, для уменьшения соответствующих рисков). Но на следующем витке может применяться каскадная модель. Если требования четко сформулированы, может применяться метод формальных преобразований.

3.3. Спецификация программного обеспечения

В этом разделе, а также в трех последующих, рассматриваются основные базовые процессы создания ПО: формирование спецификации, разработка, аттестация и модернизация программных систем. Первый из этих процессов, формирование спецификации, предназначен для определения сервисов, которыми будет обладать проектируемое ПО, а также ограничений, накладываемых на функциональные возможности и разработку программной системы. Этот процесс в настоящее время обычно называют "разработка требований" (requirements engineering). Разработка требований часто является критическим этапом в создании ПО, поскольку ошибки, допущенные на этом этапе, ведут к возникновению проблем на этапах проектирования и разработки.

Схема процесса разработки требований показана на рис. 3.8. Результатом его выполнения является разработка документации, формализующей требования, предъявляемые к системе, т.е. создание системной спецификации. В этой документации требования обычно представлены на двух уровнях детализации. На самом верхнем уровне представлены требования, определяемые конечными пользователями или заказчиками ПО; но для разработчиков необходима более детализированная системная спецификация.

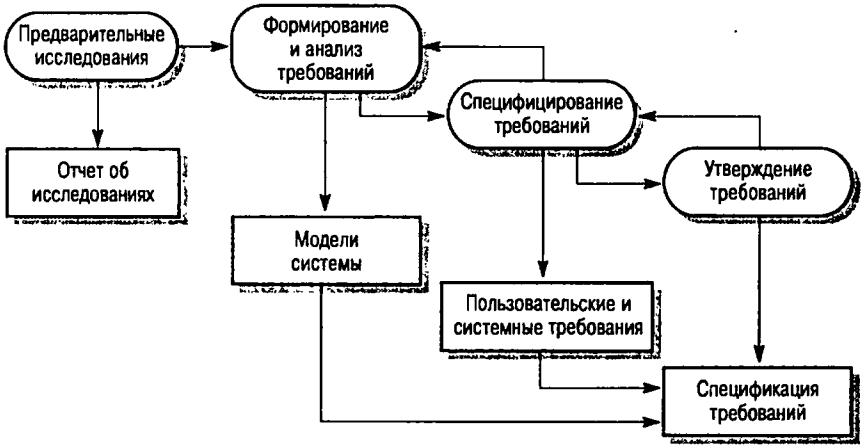


Рис. 3.8. Процесс разработки требований

Процесс разработки требований включает четыре основных этапа.

1. *Предварительные исследования.* Оценивается степень удовлетворенности пользователей существующими программными продуктами и аппаратными средствами, а также экономическая эффективность будущей системы и возможность уложиться в существующие бюджетные ограничения при ее разработке. Этот этап должен быть по возможности коротким и дешевым.
2. *Формирование и анализ требований.* Формируются системные требования путем изучения существующих аналогичных систем, обсуждения будущей системы с потенциальными пользователями и заказчиками, анализа задач, которые должна решать система, и т.п. Этот этап может включать разработку нескольких моделей системы и ее прототипов, что помогает сформировать функциональные требования к системе.
3. *Специфицирование требований.* Осуществляется перевод всей совокупности информации, собранной на предыдущем этапе, в документ, определяющий множество требований. Этот документ обычно содержит два типа требований: пользовательские — обобщенные представления заказчиков и конечных пользователей о системе; системные — детальное описание функциональных показателей системы.
4. *Утверждение требований.* Проверяется выполнимость, согласованность и полнота множества требований. В процессе формирования ограничений неизбежно возникновение каких-либо ошибок. На этом этапе они должны быть по возможности выявлены и устранены.

Конечно, процесс разработки требований трудно уложить в описанную последовательность этапов. Например, анализ требований выполняется на протяжении всего процесса их разработки, поэтому внесение новых или изменение уже сформулированных требований возможно на любом этапе. Как правило, этапы разработки требований перекрываются во времени.

3.4. Проектирование и реализация ПО

Реализация программного обеспечения — это процесс перевода системной спецификации в работоспособную систему. Этап реализации всегда включает процессы проектирования и программирования, но если для разработки ПО применяется эволюционный подход, этап реализации также может включать процесс внесения изменений в системную спецификацию.

На этапе проектирования ПО определяется его структура, данные, которые являются частью системы, интерфейсы взаимодействия системных компонентов и иногда используемые алгоритмы. Проектировщики сразу никогда не получают законченный результат — процесс проектирования обычно проходит через разработку нескольких промежуточных версий ПО. Проектирование предполагает последовательную формализацию и детализацию создаваемого ПО с возможностью внесения изменений в решения, принятые на более ранних стадиях проектирования.

Процесс проектирования может включать разработку нескольких моделей системы различных уровней обобщения. Поскольку проектирование — это процесс декомпозиции, такие модели помогают выявить ошибки, допущенные на ранних стадиях проектирования, а следовательно, позволяют внести изменения в ранее созданные модели. На рис. 3.9 показана схема процесса проектирования ПО с указанием результата каждого этапа проектирования. Эта схема построена в предположении, что все этапы процесса проектиро-

вания выполняются последовательно. На практике эти этапы перекрываются вследствие неизбежных обратных связей от одного этапа к предыдущему и повторного выполнения некоторых проектных работ.

Результатом каждого этапа проектирования является спецификация, необходимая для выполнения следующего этапа. Эта спецификация может быть абстрактной и формальной, т.е. такой, какая необходима для детализации системных требований; но она может быть и частью разрабатываемой системы. Так как процесс проектирования непрерывен, спецификации постепенно становятся все более детализированными. Конечными результатами процесса проектирования являются точные спецификации на алгоритмы и структуры данных, которые будут реализованы на следующем этапе создания ПО.

Ниже перечислены отдельные этапы процесса проектирования.

1. *Архитектурное проектирование.* Определяются и документируются подсистемы и взаимосвязи между ними.
2. *Обобщенная спецификация.* Для каждой подсистемы разрабатывается обобщенная спецификация на ее сервисы и ограничения.
3. *Проектирование интерфейсов.* Для каждой подсистемы определяется и документируется ее интерфейс. Спецификации на эти интерфейсы должны быть точно выраженными и однозначными, чтобы использование подсистем не требовало знаний о том, как они реализуют свои функции. На этом этапе можно применить методы формальных спецификаций, рассмотренные в главе 9.
4. *Компонентное проектирование.* Проводится распределение системных функций (сервисов) по различным компонентам и их интерфейсам.
5. *Проектирование структур данных.* Детально разрабатываются структуры данных, необходимые для реализации программной системы.
6. *Проектирование алгоритмов.* Детально разрабатываются алгоритмы, предназначенные для реализации системных сервисов.

Рис. 3.9. Обобщенная схема процесса проектирования

Описанная схема процесса проектирования является достаточно общей и на практике может (и должна) адаптироваться применительно к разработке конкретного программного продукта. Например, два последних этапа, проектирование структур данных и алгоритмов, могут быть как составными частями процесса проектирования, так и входить в процесс реализации ПО. Если для создания программной системы используются некоторые уже готовые компоненты, это может наложить ограничения на архитектуру системы и интерфейсы системных модулей. Это означает, что количество компонентов, требующих проектирования, значительно уменьшится. Если в процессе проектирования используется метод проб и ошибок, то системные интерфейсы могут разрабатываться после определения структур данных.

3.4.1. Методы проектирования

Во многих проектах разработки ПО процесс проектирования выполняется с помощью *специально подобранных* методов. Отталкиваясь от множества требований, обычно записанных естественным языком, сначала выполняется неформальное проектирование. Комментарии к программному коду и промежуточные спецификации могут изменяться в процессе реализации системы. После завершения стадии реализации (т.е. программирования и отладки системы) в проектную документацию также вносятся изменения, призванные устранить ошибки и неполноту описания системы в первоначальной спецификации.

Наиболее разработанным подходом к проектированию ПО обладают так называемые структурные методы, которые предлагают множество формализованных нотаций и нормативных руководств для проектирования программных продуктов. В качестве примера этих методов можно назвать структурное проектирование [79, 7*, 24*], структурный анализ систем [125, 10*, 12*], разработку систем Джексона (Jackson, [181, 9*]), а также разнообразные методы, основанные на объектно-ориентированном подходе [295, 54, 302, 55, 303, 304, 7*, 32*, 34*].

Применение структурных методов обычно приводит к созданию графических моделей системы и большому объему проектной документации. CASE-средства (см. раздел 3.7) предназначены для поддержки именно таких методов. Структурные методы успешно применялись во многих программных проектах. Они значительно снижают стоимость разработки, поскольку используют стандартные нотации для получения стандартной проектной документации. Ни об одном из этих методов нельзя сказать, что он лучше или хуже других. Успешное или неуспешное применение того или иного метода часто зависит от типа разрабатываемого ПО.

Каждый структурный метод включает такие компоненты, как модель процесса проектирования, стандартизованные нотации для представления структуры системы, форматы отчетов, правила и нормативные указания по проектированию. Хотя разработано большое количество таких методов, они имеют нечто общее. Структурные методы поддерживают все или по крайней мере некоторые из перечисленных ниже моделей систем.

1. Модель потоков данных, где система моделируется в виде потока данных, преобразуемых в этой системе.
2. Модель "сущность–связь", которая применяется для описания сущностей (объектов программной системы) и связей между ними. Эта модель часто используется при проектировании структур баз данных.
3. Структурная модель, предназначенная для документирования системных компонентов и их взаимосвязей.

4. Объектно-ориентированные методы, с помощью которых получают иерархическую модель системы, модели статических и динамических отношений между объектами и модель взаимодействия объектов во время работы системы.

Некоторые структурные методы дополняются другими системными моделями, такими как диаграммы переходов (из одного состояния в другое) или сценарии жизни сущностей, которые показывают последовательность преобразований для каждой сущности. Многие методы предполагают наличие централизованных хранилищ (репозиториев) для системной информации или словарей используемых данных. Многие из этих подходов к моделированию систем описаны в главе 7.

На практике методы представляют нормативные руководства неформально, так что различные проектировщики могут реализовать разные пути проектирования. Фактически эти "методы" являются набором стандартных нотаций и просто отображают успешную практику проектирования. Следуя этим методам и их нормативным руководствам, можно прийти к рациональному и разумному процессу проектирования. Вместе с тем творчество проектировщиков должно проявиться в способе декомпозиции системы, адекватно отображающей системные требования. С другой стороны, проведенные исследования труда проектировщиков показали, что чаще всего они просто слепо следуют этим методам [20]. Да и сами методы они выбирают в зависимости от частных обстоятельств, а не в соответствии с их достоинствами или недостатками.

3.4.2. Программирование и отладка

Процесс программирования (написания программного кода, кодирования) обычно следует непосредственно за процессом проектирования. Но для некоторых классов программ, например критических по надежности систем, последняя стадия проектирования (детальное проектирование) и начало кодирования могут перекрываться. В процессе проектирования могут использоваться CASE-средства, которые позволяют получить скелетную программу. Такая программа содержит код для определения и реализации интерфейсов, и во многих случаях программисту остается только добавить код, реализующий некоторые детали функционирования программного компонента.

Программирование – индивидуальный процесс, здесь не существует общих правил, которым необходимо следовать при написании программного кода. Некоторые программисты начинают кодирование с компонентов, которые они хорошо понимают, оставляя напоследок кодирование компонентов, которые являются для них "темными". Другие применяют противоположный подход, оставляя простые для них компоненты на потом.

Обычно программисты сами тестируют написанный ими программный код для обнаружения возможных ошибок и программных дефектов. Этот процесс называется *отладкой* программы. В принципе тестирование и отладка являются разными процессами. При тестировании устанавливается наличие программных ошибок. В ходе отладки устанавливается местоположение ошибок, затем они устраняются. На рис. 3.10 показан возможный процесс отладки программы. Отладка может быть частью как процесса разработки, так и процесса тестирования ПО.

Проводящий отладку программист должен сгенерировать такие режимы работы системы, которые помогут обнаружить программные ошибки по аномальному поведению системы. Локализация ошибок может потребовать проведения ручной трассировки кода

программы. В процессе тестирования и отладки могут помочь отладочные средства, показывающие значения программных переменных и выполняющие трассировку исполняемых операторов.

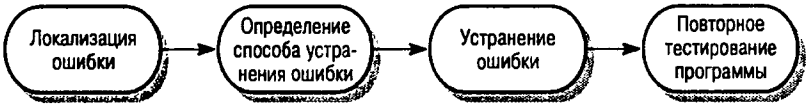


Рис. 3.10. Процесс отладки

3.5. Аттестация программных систем

Аттестация ПО, или более обобщенно – верификация и аттестация, предназначена показать соответствие системы ее спецификации, а также ожиданиям и требованиям заказчика и пользователей. К процессу аттестации также можно отнести элементы контроля, такие как инспекция и оценивание (см. главу 19), которые выполняются на каждом этапе создания ПО – от формирования общих требований до кодирования программ. Но все-таки основные действия по аттестации выполняются после завершения реализации на этапе тестирования законченной системы (глава 20).

За исключением небольших программ, программные системы невозможно протестировать как единый целый программный элемент. Большие системы строятся на основе подсистем, которые, в свою очередь, строятся из модулей, модули же компонуются из программ-процедур и программ-функций. Для таких систем процесс тестирования выполняется постепенно, по мере реализации системы.

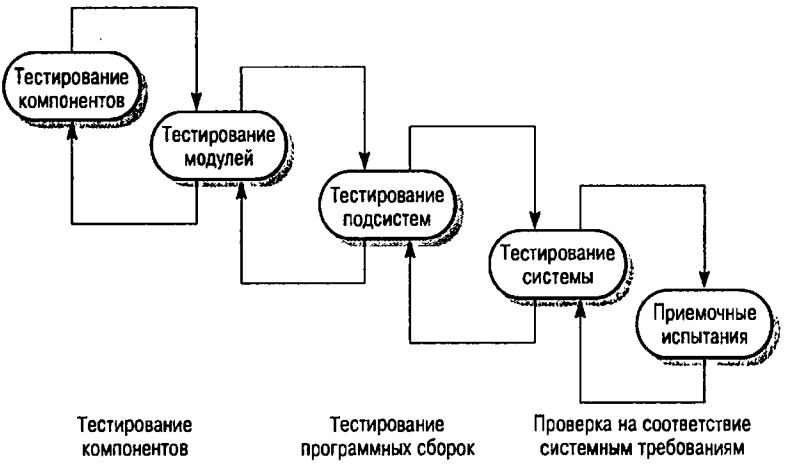


Рис. 3.11. Процесс тестирования

На рис. 3.11 показан пятиэтапный процесс тестирования, где сначала тестируются отдельные программные компоненты и подсистемы, затем собранная система и наконец система с данными, предоставляемыми заказчиком. В идеале ошибки в программных компонентах должны обнаруживаться и исправляться еще в процессе их кодирования, а ошибки и упущения в интерфейсах – во время сборки системы. Но, поскольку после обнаружения любых программных ошибок необходимо выполнить отладку программы, это

приводит к необходимости повторения некоторых этапов тестирования. Например, если программная ошибка проявилась на этапе сборки системы, необходимо повторить процесс тестирования того программного компонента, в котором обнаружена эта ошибка. Поэтому процесс тестирования итерационный, с обратной передачей информации с последующих этапов на предыдущие.

Процесс тестирования состоит из нескольких этапов.

1. *Тестирование компонентов.* Тестируются отдельные компоненты для проверки правильности их функционирования. Каждый компонент тестируется независимо от других.
2. *Тестирование модулей.* Программный модуль – это совокупность зависимых компонентов, таких как описание класса объектов, декларирование абстрактных типов данных и набор процедур и функций. Каждый модуль тестируется независимо от других системных модулей.
3. *Тестирование подсистем.* Тестируются наборы модулей, которые составляют отдельные подсистемы. Основная проблема, которая часто проявляется на этом этапе, – несогласованность модульных интерфейсов. Поэтому при тестировании подсистем основное внимание уделяется обнаружению ошибок в модульных интерфейсах путем прогона их через все возможные режимы.
4. *Тестирование системы.* Из подсистем собирается конечная система. На этом этапе основное внимание уделяется совместимости интерфейсов подсистем и обнаружению программных ошибок, которые проявляются в виде непредсказуемого взаимодействия между подсистемами. Здесь также проводится аттестация системы, т.е. проверяется соответствие системной спецификации ее функциональных и нефункциональных показателей, а также оцениваются интегрированные характеристики системы.
5. *Приемочные испытания.* Это конечный этап процесса тестирования, после которого система принимается к эксплуатации. Здесь система тестируется с привлечением данных, предоставляемых заказчиком системы, а не на основе тестовых данных, как было на предыдущем этапе. На этом этапе могут проявиться ошибки, допущенные еще на этапе определения системных требований, поскольку испытания с реальными данными могут дать иной результат, чем тестирование со специально подобранными тестовыми данными. Приемочные испытания могут также выявить другие проблемы в системных требованиях, если реальные системные характеристики не отвечают потребностям заказчика или система функционирует непредвиденным образом.

Тестирование программных компонентов и модулей обычно выполняется тем программистом, который их разрабатывал. Программисты имеют собственные наборы тестовых данных и тестируют программный код постепенно, по мере его создания. Такой подход к тестированию отдельных компонентов и модулей вполне оправдан, поскольку никто лучше программиста, разработавшего программный компонент, его не знает, и поэтому он может подобрать наилучшие тестовые данные. Тестирование программных элементов можно рассматривать как часть процесса их создания, поэтому мы вправе ожидать точного соответствия этих элементов и их спецификаций.

Последние этапы тестирования выполняются в процессе сборки системы, к которой привлекается несколько программистов. Поэтому эти работы должны быть спланированы

заранее. Если тестирование выполняет независимая команда испытателей, планы проведения тестирования должны быть согласованы с этапами разработки спецификации и проектирования. На рис. 3.12 показано, как планы тестирования могут быть связаны с другими процессами разработки ПО.

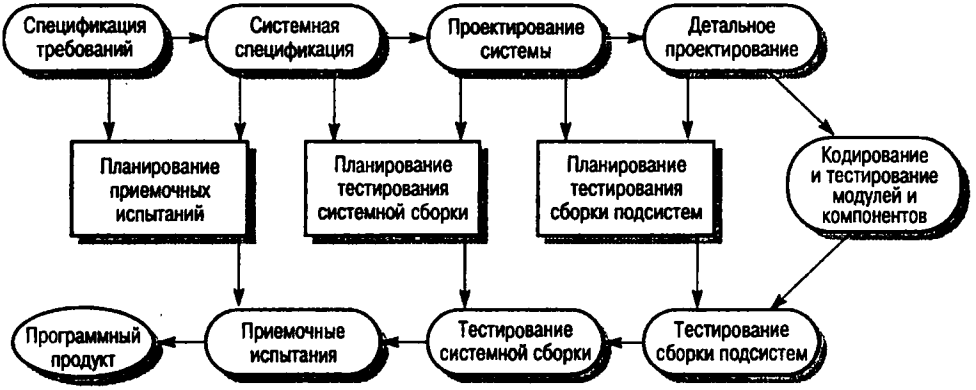


Рис. 3.12. Этапы тестирования в процессе разработки ПО

Приемочные испытания иногда называют *альфа-тестированием*. Сделанные на заказ системы предназначены для одного заказчика. Для таких систем процесс альфа-тестирования продолжается до тех пор, пока разработчики и заказчик не удостоверятся в том, что разработанная система полностью соответствует системным требованиям.

Если система разрабатывается для продажи на рынке программных продуктов, используется так называемое *бета-тестирование*. Для бета-тестирования система рассылается большому числу потенциальных пользователей и заказчиков. Они отсылают разработчикам отчеты о выявленных проблемах в эксплуатации системы. Бета-тестирование позволяет проверить систему в реальных условиях эксплуатации и найти ошибки, пропущенные разработчиками. После получения отчетов об испытаниях система модернизируется и снова передается на бета-тестирование либо сразу поступает в продажу.

3.6. Эволюция программных систем

Одна из основных причин того, что в настоящее время в большие сложные системы все шире внедряется программное обеспечение, заключается в гибкости программных систем. После принятия решения о разработке и производстве аппаратных компонентов системы внесение в них изменений становится весьма дорогостоящим. С другой стороны, в программное обеспечение можно вносить изменения в течение всего процесса разработки системы. Эти изменения также могут быть крайне дорогостоящими, но все-таки они значительно дешевле изменений в аппаратном оборудовании системы.

Исторически сложилось так, что существует четкая "демаркационная линия" между процессом разработки системы и процессом ее совершенствования, точнее, процессом сопровождения системы. Разработка системы рассматривается как творческий процесс, начиная с этапа выработки общей концепции системы и заканчивая получением работающего программного продукта. Сопровождение системы – это внесение изменений в систему, которая уже находится в эксплуатации. И хотя стоимость сопровождения может в несколько раз превышать стоимость разработки, все равно процесс сопро-

вождения считается менее творческим и ответственным, чем процесс первоначального создания системы.

В настоящее время упомянутая демаркационная линия между процессами разработки и сопровождения постепенно стирается. Только немногие вновь созданные программные системы можно назвать полностью новыми. Поэтому имеет смысл рассматривать процесс сопровождения как непрерывное продолжение процесса разработки. Вместо двух отдельных процессов рациональнее принять эволюционный подход инженерии программного обеспечения, где программные продукты в течение своего жизненного цикла непрерывно изменяются (эволюционируют) в ответ на изменения в системных требованиях и потребностях пользователей. Схема этого эволюционного процесса программных систем показана на рис. 3.13.

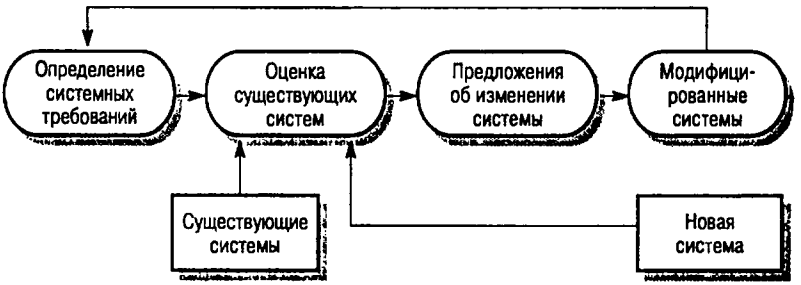


Рис. 3.13. Эволюция систем

3.7. Автоматизированные средства разработки ПО

Аббревиатура CASE (Computer-aided Software Engineering – автоматизированная разработка ПО) обозначает специальный тип программного обеспечения, предназначенного для поддержки таких процессов создания ПО, как разработка требований, проектирование, кодирование и тестирование программ. Поэтому к CASE-средствам относятся редакторы проектов, словари данных, компиляторы, отладчики, средства построения систем и т.п.

CASE-технологии предлагают поддержку процесса создания ПО путем автоматизации некоторых этапов разработки, а также создания и предоставления информации, необходимой для разработки. Приведем примеры тех процессов, которые можно автоматизировать с помощью CASE-средств.

1. Разработка графических моделей системы на этапах создания спецификации и проектирования.
2. Проектирование структуры ПО с использованием словарей данных, хранящих информацию об объектах структуры и связях между ними.
3. Генерирование пользовательских интерфейсов на основе графического описания интерфейса, создаваемого в диалоговом режиме.
4. Отладка программ на основе информации, получаемой в ходе выполнения программы.
5. Автоматическая трансляция программ, написанных на устаревших языках программирования (например, COBOL), в программы, написанные на современных языках.

В настоящее время подходящие CASE-технологии существуют для большинства процессов, выполняемых в ходе разработки ПО. Это ведет к определенному улучшению качества создаваемых программ и повышению производительности труда разработчиков программного обеспечения. Вместе с тем эти достижения значительно уступают тем ожиданиям, которые присутствовали при зарождении CASE-технологий. Тогда считалось, что стоит только внедрить CASE-средства — и можно получить весьма значительное повышение и качества программ, и производительности труда. Фактически это повышение составляет примерно 40% [165]. Хотя и это повышение весьма значительно, CASE-технологии не совершили революции в инженерии программного обеспечения, как ожидалось.

Расширение применения CASE-технологий ограничивают два фактора.

1. Создание ПО, особенно этап проектирования, во многом является творческим процессом. Существующие CASE-средства автоматизируют рутинные процессы, попытки привлечь их к решению интеллектуальных и творческих задач проектирования особым успехом не увенчались.
2. Во многих организациях-разработчиках создание ПО — результат работы команды специалистов по программному обеспечению. При этом много времени тратится на "пустое" общение между членами команды разработчиков. В этой ситуации CASE-технологии не могут предложить ничего такого, что способно повысить производительность труда разработчиков.

Отомрут ли эти факторы в будущем, пока неясно. Я считаю маловероятным появление CASE-технологий, поддерживающих творческие элементы процесса проектирования систем и коллективный труд команды разработчиков. Однако системы поддержки общего процесса проектирования и групповой работы существуют и используются в процессе создания ПО.

В настоящее время сложилась развитая индустрия CASE-средств, круг возможных поставщиков и разработчиков этих программных продуктов очень широк[3] . Вместо того чтобы детально рассматривать отдельные CASE-продукты, здесь будет сделан их обзор, а в последующих главах книги при рассмотрении соответствующих тем некоторые из них будут описаны более подробно. На Web-странице этой книги имеются ссылки на более подробный материал о CASE-технологиях и на поставщиков CASE-средств.

3.7.1. Классификация CASE-средств

Классификация CASE-средств помогает понять их основные типы и роль, которую они играют в поддержке процессов создания программного обеспечения. Существует несколько различных классификаций CASE-средств, и каждая предлагает свой взгляд на эти программные продукты. В этом разделе рассматриваются следующие классификации.

1. Классификация *по выполняемым функциям*.
2. Классификация *по типам процессов разработки*, которые они поддерживают.
3. Классификация *по категориям*, где CASE-средства классифицируются по степени интеграции программных модулей, поддерживающих различные процессы разработки.

В табл. 3.1 представлена классификация по выполняемым функциям с примерами соответствующих CASE-средств. Это неполный список типов CASE-средств, в частности здесь не представлены средства поддержки повторного использования программных компонентов.

---

[3] *Список фирм, поставляющих CASE-средства на российский рынок, можно найти в [6*].* – Прим. ред.

Таблица 3.1. Классификация CASE-средств по выполняемым функциям

Тип CASE-средства	Примеры
Средства планирования	Средства системы PERT[4], средства оценивания, электронные таблицы
Средства редактирования	Текстовые редакторы, редакторы диаграмм, тестовые процессоры
Средства управления изменениями	Средства оперативного контроля за требованиями, системы управления изменениями
Средства управления конфигурацией[5]	Системы управления версиями ПО, средства построения систем
Средства прототипирования	Языки программирования самого высокого уровня, генераторы пользовательских интерфейсов
Средства, ориентированные на поддержку определенных методов	Редакторы системных структур, словари данных, генераторы программного кода
Средства, ориентированные на определенные языки программирования	Компиляторы, интерпретаторы
Средства анализа программ	Генераторы перекрестных ссылок, статические и динамические анализаторы программ
Средства тестирования	Генераторы тестовых данных, компараторы[6] файлов
Средства отладки	Интерактивные средства отладки
Средства документирования	Программы разметки страниц, редакторы изображений, генераторы отчетов
Средства модернизации ПО	Системы создания перекрестных ссылок, системы модернизации программ

В табл. 3.2 представлена другая классификация CASE-средств. Классификация по типам показывает, какие процессы создания ПО поддерживаются теми или иными CASE-средствами. Средства планирования и оценивания, редактирования текстов, подготовки документации и управления конфигурацией можно использовать на всех этапах разработки ПО.

---

[4] *PERT (Program Evaluation and Review Technique) – известная система планирования и руководства разработками программных систем.* – Прим. ред.

[5] *Конфигурацией ПО называется совокупность его функциональных характеристик и физических показателей, зафиксированная в системной спецификации.* – Прим. ред.

[6] *Компараторы – специальные программы сравнения каких-либо объектов. В данном случае имеются в виду программы сравнения файлов, содержащих программный код.* – Прим. ред.

Таблица 3.2. Классификация CASE-средств по типам поддерживаемых ими процессов разработки

	Специфицирование	Проектирование	Реализация	Аттестация
Средства модернизации ПО			•	
Средства тестирования			•	•
Средства отладки			•	•
Средства анализа программ			•	•
Средства, ориентированные на определенные языки программирования		•	•	
Средства, ориентированные на поддержку определенных методов	•	•		
Средства протопипирования	•			•
Средства управления конфигурацией		•	•	
Средства управления изменениями	•	•	•	•
Средства документирования	•	•	•	•
Средства редактирования	•	•	•	•
Средства планирования	•	•	•	•

Другая классификация CASE-средств строится на основе широты охвата процессов разработки ПО, поддерживаемых данным средством. В статье [120] предложена классификация, содержащая следующие три категории[7].

1. *Вспомогательные программы* (tools) поддерживают отдельные процессы разработки ПО, такие как проверка непротиворечивости архитектуры системы, компиляция программ, сравнение результатов тестов и т.п. Вспомогательные программы могут быть универсальными функционально-законченными средствами (например, текстовой процессор) или могут входить в состав инструментальных средств.
2. *Инструментальные средства* (workbenches) поддерживают определенные процессы разработки ПО, например создание спецификации, проектирование и т.д. Обычно инструментальные средства являются набором вспомогательных программ, которые в большей или меньшей степени интегрированы.

[7] *В литературе по CASE-технологиям можно встретить и другую классификацию CASE-средств по категориям: вспомогательные программы (tools), инструментальные пакеты разработчика (toolkits) и автоматизированные рабочие места разработчика (workbenches). По существу, эта классификация совпадает с приведенной в данной книге, различия только в названиях категорий.* – Прим. ред.

3. *Рабочие среды разработчика* (environments) поддерживают все или большинство процессов разработки ПО. Рабочие среды обычно включают несколько различных интегрированных инструментальных средств.

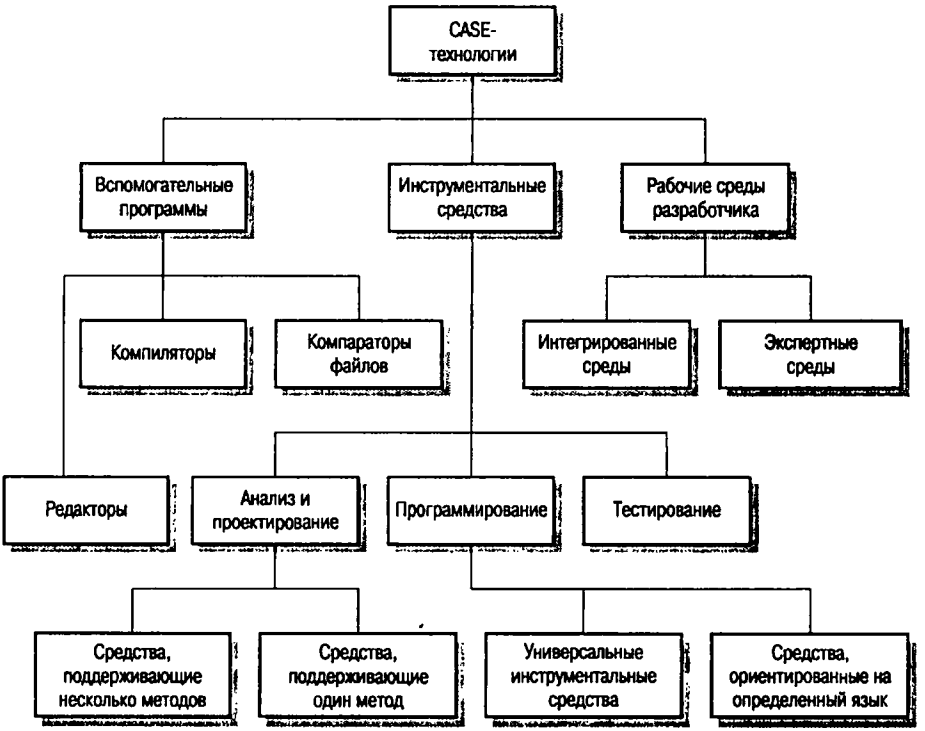


Рис. 3.14. Классификация CASE-средств по категориям

На рис. 3.14 схематично представлена классификация по категориям с примерами CASE-средств разных категорий. Разумеется, на одной схеме невозможно показать все типы вспомогательных программ и инструментальных средств, многие из них здесь не представлены.

Необходимые отдельные вспомогательные программы выбираются разработчиком ПО обычно по своему усмотрению. Инструментальные средства, как правило, поддерживают определенные методы разработки в соответствии с некоторой моделью процесса создания ПО и содержат наборы правил и нормативных указаний, которыми следует руководствоваться в процессе разработки. Рабочие среды разработчика я разделил на интегрированные и экспертные. Интегрированные рабочие среды предоставляют инфраструктуру поддержки для данных, управления и интеграции системных представлений. Экспертные рабочие среды более интеллектуальны. Они включают базу знаний о процессах создания ПО и механизм, который в соответствии с выбранной моделью процесса создания ПО предлагает разработчику для применения те или иные вспомогательные программы и инструментальные средства.

На практике границы между CASE-средствами разных категорий размыты. Вспомогательную программу можно приобрести как отдельный продукт, но она может использоваться для поддержки различных процессов разработки. Например, большинство текстовых про-

цессоров в настоящее время располагают встроенными редакторами диаграмм; или инструментальные CASE-средства для проектирования все чаще предлагают поддержку процессам программирования и тестирования, тем самым приближаясь к рабочим средам. Поэтому не всегда можно легко позиционировать какой-либо CASE-продукт по категориям в соответствии с этой классификацией. Вместе с тем классификация по категориям полезна для понимания того, насколько широк диапазон процессов разработки, которые могут быть поддержаны тем или иным CASE-средством.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Процесс создания программного обеспечения — это совокупность процессов, выполняемых при разработке программных продуктов. Модели процесса создания ПО — абстрактные представления этих процессов.
- Любой процесс создания программного обеспечения включает этапы разработки системной спецификации, проектирования и реализации, аттестации и модернизации ПО.
- Обобщенные модели создания ПО описывают организацию процесса разработки программных систем. К таким моделям относятся: каскадная модель, эволюционная модель разработки, модель формальной разработки систем и модель разработки ПО на основе ранее созданных компонентов.
- Итерационные модели разработки ПО представляют процесс создания программных систем в виде повторяющихся циклов определенных этапов разработки. Достоинством данного подхода является возможность избежать преждевременного и до конца не продуманного утверждения системной спецификации и результатов проектирования. Примерами итерационных моделей служат модель пошаговой разработки и спиральная модель.
- Определение требований — это процесс разработки системной спецификации.
- Проектирование и реализация — это процессы преобразования системной спецификации в систему исполняемых программ.
- Аттестация программного обеспечения — процесс проверки соответствия разработанной системы ее спецификации и потребностям пользователей.
- Эволюция программного обеспечения — это модернизация существующих программных систем в соответствии с новыми требованиями. В настоящее время этот процесс становится одним из этапов разработки небольших и среднего размера программных систем.
- CASE-технологии обеспечивают автоматизированную поддержку процессов создания ПО. Вспомогательные CASE-программы поддерживают отдельные процессы разработки; инструментальные CASE-средства поддерживают некоторое множество взаимосвязанных процессов разработки; рабочие CASE-среды обеспечивают поддержку всех или большинства процессов, выполняемых при создании ПО.

Упражнения

3.1. Предложите подходящую модель процесса создания ПО для разработки перечисленных ниже программных систем. Обоснуйте свое предложение.

3.2. Система управления торможением автомобиля.

3.3. Система поддержки процесса сопровождения программного обеспечения.

3.4. Университетская система учета и отчетности, которая должна заменить существующую систему.

3.5. Интерактивная система просмотра железнодорожных расписаний для пассажиров.

3.6. Поясните, почему программы, создаваемые в соответствии с эволюционной моделью разработки, трудны для сопровождения.

3.7. Объясните, как каскадную модель и эволюционную модель с прототипированием можно объединить со спиральной моделью разработки ПО.

3.8. Объясните, почему в процессе определения требований необходимо различать разработку пользовательских требований и разработку системных требований.

3.9. Опишите основные этапы процесса проектирования ПО, укажите выходной результат каждого этапа. С помощью диаграммы "сущность-связь" покажите возможные взаимосвязи между выходными результатами разных этапов процесса проектирования.

3.10. Назовите пять основных компонентов любых методов проектирования. Какие методы проектирования вы знаете? Опишите их компоненты. Оцените полноту этих методов.

3.11. Разработайте модель процесса тестирования исполняемой программы.

3.12. Покажите, как классификация CASE-технологий может помочь менеджеру в процессе приобретения CASE-систем.

3.13. Опишите CASE-средства, которые можно использовать в вашей рабочей среде разработчика, и классифицируйте их по нескольким параметрам (выполняемая функция, поддерживаемый процесс, количество поддерживаемых процессов).

3.14. Известно, что внедрение любой новой технологии значительно влияет на рынок рабочей силы и (по крайней мере временно) увеличивает уровень безработицы. Обсудите возможные последствия внедрения CASE-технологий для специалистов по программному обеспечению. Уменьшат ли эти технологии занятость разработчиков ПО? Если вы считаете, что уменьшат, то этичны ли действия специалистов, внедряющих CASE-технологии?

4 Управление проектами

Цели

Цель настоящей главы — обзор процессов управлени
программными проектами. Прочитав эту главу, в
должны:

- понимать различия между управлением про
граммными проектами и управлением инженер
ными проектами других типов;
- знать основные задачи, стоящие перед руководи
телем программного проекта;
- понимать значимость и роль этапа планировани
проекта среди всех этапов процесса создани
ПО;
- знать графические способы представления гра
фиков работ;
- иметь представление о типах рисков, возникаю
щих при реализации программных проектов,
процессы управления этими рисками.

Содержание

Проблемы управления программными проектами впервые проявились в 60-х – начале 70-х годов, когда провалились многие большие проекты по разработке программных продуктов. Были зафиксированы задержки в создании ПО, оно было ненадежным, затраты на разработку в несколько раз превосходили первоначальные оценки, созданные программные системы часто имели низкие показатели производительности [60]. Провалы этих проектов обусловливались не только некомпетентностью руководителей и программистов. Напротив, в этих больших поисковых проектах принимали участие люди, уровень квалификации которых был явно выше среднего. Причины провалов коренились в тех подходах, которые использовались в управлении проектами. Применяемая методика была основана на опыте управления техническими проектами и оказалась неэффективной при разработке программного обеспечения.

Здесь важно понять разницу между профессиональной разработкой ПО и любительским программированием. Необходимость управления программными проектами вытекает из того "прискорбного" факта, что процесс создания профессионального ПО всегда является субъектом бюджетной политики организации, где оно разрабатывается, и имеет временные ограничения. Работа руководителя программного проекта по большому счету заключается в том, чтобы гарантировать выполнение этих бюджетных и временных ограничений с учетом бизнес-целей организации относительно разрабатываемого ПО.

Менеджеры проектов призваны спланировать все этапы разработки программного продукта. Они также должны контролировать ход выполнения работ и соблюдения всех требуемых стандартов. Постоянный контроль за ходом выполнения работ необходим для того, чтобы процесс разработки не выходил за временные и бюджетные ограничения. Хорошее управление не гарантирует успешного завершения проекта, но плохое управление обязательно приведет к его провалу. Это может выразиться в задержке сроков сдачи готового ПО, в превышении сметной стоимости проекта и в несоответствии готового ПО спецификации требований.

Руководители программных проектов выполняют такую же работу, что и руководители технических проектов. Вместе с тем процесс разработки ПО существенно отличается от процессов реализации технических проектов, что порождает определенные сложности в управлении программными проектами. Приведем небольшой список этих отличий.

1. *Программный продукт нематериален.* Менеджер судостроительного проекта или проекта постройки здания видит результаты выполнения своего проекта. Если реализация проекта отстает от графика, это также видно воочию, так как часть конструкции не завершена. В противоположность этому программное обеспечение нематериально. Его нельзя увидеть или потрогать. Менеджер программного проекта не видит процесс "роста" разрабатываемого ПО. Он может полагаться только на документацию, которая фиксирует процесс разработки программного продукта.
2. *Не существует стандартных процессов разработки ПО.* На сегодняшний деиь не существует четкой зависимости между процессом создания ПО и типом создаваемого программного продукта. Другие технические дисциплины имеют длительную историю, процессы разработки технических изделий многократно опробованы и проверены. Процессы создания большинства технических систем хорошо изучены. Изучением же процессов создания ПО специалисты занимаются только несколько последних лет. Поэтому пока нельзя точно предсказать, на каком этапе процесса разработки ПО могут возникнуть проблемы, угрожающие всему программному проекту.
3. *Большие программные проекты – это часто "одноразовые" проекты.* Большие программные проекты, как правило, значительно отличаются от проектов, реализованных ранее. Поэтому, чтобы уменьшить неопределенность в планировании проекта, руководители проектов должны обладать очень большим практическим опытом. Но постоянные технологические изменения в компьютерной технике и коммуникационном оборудовании обесценивают предыдущий опыт. Знания и навыки, накопленные опытом, могут не востребоваться в новом проекте.

Перечисленное выше может привести к тому, что реализация проекта выйдет из временного графика или превысит бюджетные ассигнования. Программные системы зачастую оказываются новинками как в "идеологическом", так и в техническом плане. Технические проекты, которые являются инновационными (например, новая транспортная система), также часто нарушают временные графики работ. Поэтому, предвидя возможные проблемы в реализации программного проекта, следует всегда помнить, что многим из них свойственно выходить за рамки временных и бюджетных ограничений.

Управление программными проектами — тема весьма обширная, которую невозможно осветить в одной главе. Поэтому в настоящей главе дается только введение в эту тему и рассматривается три основных процесса, выполняемых в рамках управления проектами, а именно: планирование проекта, составление графика работ и управление рисками. В части VI представлены другие аспекты управления проектами разработки ПО, в том числе управление персоналом, оценивание стоимости проекта и управление качеством ПО.

4.1. Процессы управления

Невозможно описать и стандартизировать все работы, выполняемые менеджером проекта по созданию ПО. Эти работы весьма существенно зависят от организации, где выполняется разработка ПО, и от типа создаваемого программного продукта. Но в любом случае большинство менеджеров ответственны за выполнение всех или некоторых из приведенных ниже процессов управления.

- Написание предложений по созданию ПО.
- Планирование и составление графика работ по созданию ПО.
- Оценивание стоимости проекта.
- Контроль за ходом выполнения работ.
- Подбор персонала.
- Написание отчетов и представлений.

Первая стадия программного проекта может состоять из написания предложений по реализации этого проекта. Предложения должны содержать описание целей проектов и способов их достижения. Они также обычно включают в себя оценки финансовых и временных затрат на выполнение проекта. При необходимости здесь могут приводиться обоснования для передачи проекта на выполнение сторонней организации или команде разработчиков.

Написание предложений — очень ответственная работа, так как для многих организаций вопрос о том, будет ли проект выполняться самой организацией или разрабатываться по контракту сторонней компанией, является критическим. Невозможно дать каких-либо рекомендаций по написанию предложений, многое здесь зависит от опыта менеджера. Эйрон (Aron) [12] считает эту работу менеджера одной из важнейших среди других выполняемых им работ.

На этапе планирования проекта определяются процессы, этапы и полученные на каждом из них результаты, которые должны привести к выполнению проекта. Реализация этого плана приведет к достижению целей проекта. Определение стоимости проекта напрямую связано с его планированием, поскольку здесь оцениваются ресурсы, требующиеся для выполнения плана. Эти вопросы обсуждаются далее в этой главе, а также в главе 23.

Контроль за ходом выполнения работ (мониторинг проекта) – это непрерывный процесс, продолжающийся в течение всего срока реализации проекта. Менеджер должен постоянно отслеживать ход реализации проекта и сравнивать фактические и плановые показатели выполнения работ с их стоимостью. Хотя многие организации имеют механизмы формального мониторинга работ, опытный менеджер может составить ясную картину о стадии развитии проекта просто путем неформального общения с разработчиками.

Неформальный мониторинг часто помогает обнаружить потенциальные проблемы, которые в явном виде могут обнаружиться позднее. Например, ежедневное обсуждение хода выполнения работ может выявить отдельные недоработки в создаваемом программном продукте. Вместо ожидания отчетов, в которых будет отражен факт "пробуксовки" графика работ, менеджер может обсудить со специалистами намечающиеся программистские проблемы и не допустить срыва графика работ.

В течение реализации проекта обычно происходит несколько формальных контрольных проверок хода выполнения работ по созданию ПО. Такие проверки должны дать общую картину хода реализации проекта в целом и показать, насколько уже разработанная часть ПО соответствует целям проекта.

Время выполнения больших программных проектов может занимать несколько лет. В течение этого времени цели и намерения организации, заказавшей программный проект, могут существенно измениться. Может оказаться, что разрабатываемый программный продукт стал уже ненужным либо исходные требования к создаваемому ПО просто устарели и их необходимо кардинально менять. В такой ситуации руководство организации-разработчика может принять решение о прекращении разработки ПО или об изменении проекта в целом с тем, чтобы учесть изменившиеся цели и намерения организации-заказчика.

Руководители – менеджеры проектов обычно обязаны сами подбирать исполнителей для своих проектов. В идеальном случае профессиональный уровень исполнителей должен соответствовать той работе, которую они будут выполнять в ходе реализации проекта. Однако во многих случаях менеджеры должны полагаться на команду разработчиков, которая далека от идеальной. Такая ситуация может быть вызвана следующими причинами.

1. Бюджет проекта не позволяет привлечь высококвалифицированный персонал. В таком случае за меньшую плату привлекаются менее квалифицированные специалисты.
2. Бывают ситуации, когда невозможно найти специалистов необходимой квалификации как в самой организации-разработчике, так и вне ее. Например, в организации "лучшие люди" могут быть уже заняты в других проектах.
3. Организация хочет повысить профессиональный уровень своих работников. В этом случае она может привлечь к участию в проекте неопытных или недостаточно квалифицированных работников, чтобы они приобрели необходимый опыт и поучились у более опытных специалистов.

Таким образом, почти всегда подбор специалистов для выполнения проекта имеет определенные ограничения и не является свободным. Вместе с тем необходимо, чтобы хотя бы несколько членов группы разработчиков имели квалификацию и опыт, достаточные для работы над данным проектом. В противном случае невозможно избежать ошибок в разработке ПО. Проблемы создания команд разработчиков и подбора персонала более подробно рассматриваются в главе 22.

Менеджер проекта обычно обязан посылать отчеты о ходе его выполнения как заказчику, так и подрядным организациям. Это должны быть краткие документы, основанные на информации, извлекаемой из подробных отчетов о проекте. В этих отчетах должна быть та информация, которая позволяет четко оценить степень готовности создаваемого программного продукта.

4.2. Планирование проекта

Эффективное управление программным проектом напрямую зависит от правильного планирования работ, необходимых для его выполнения. План помогает менеджеру предвидеть проблемы, которые могут возникнуть на каких-либо этапах создания ПО, и разработать превентивные меры для их предупреждения или решения. План, разработанный на начальном этапе проекта, рассматривается всеми его участниками как руководящий документ, выполнение которого должно привести к успешному завершению проекта. Этот первоначальный план должен максимально подробно описывать все этапы реализации проекта.

Структура плана создания ПО рассматривается в разделе 4.2.1. Здесь лишь отметим, что, кроме разработки плана проекта, на менеджера ложится обязанность разработки других видов планов. Эти виды планов кратко описаны в табл. 4.1 и подробно обсуждаются в соответствующих главах книги.

Таблица 4.1. Виды планов

План	Описание
План качества	Описывает стандарты и мероприятия по поддержке качества разрабатываемого ПО (глава 24)
План аттестации	Описывает способы, ресурсы и перечень работ, необходимых для аттестации программной системы (глава 19)
План управления конфигурацией	Описывает структуру и процессы управления конфигурацией (глава 29)
План сопровождения ПО	Предлагает план мероприятий, требующихся для сопровождения ПО в процессе его эксплуатации, а также расчет стоимости сопровождения и необходимые для этого ресурсы (глава 27)
План по управлению персоналом	Описывает мероприятия, направленные на повышение квалификации членов команды разработчиков (глава 22)

В листинге 4.1 показан процесс планирования создания ПО в виде псевдокода. Здесь сделан акцент на том, что планирование — это итерационный процесс. Поскольку в процессе выполнения проекта постоянно поступает новая информация, план должен регулярно пересматриваться. Важными факторами, которые должны учитываться при разработке плана, являются финансовые и деловые обязательства организации. Если они изменяются, эти изменения также должны найти отражение в плане работ.

Листинг 4.1. Процесс планирования проекта

```
Определение проектных ограничений
Первоначальная оценка параметров проекта
Определение этапов выполнения проекта и контрольных отметок
```

```
while пока проект не завершится или не будет остановлен loop
    Составление графика работ
    Начало выполнения работ
    Ожидание окончания очередного этапа работ
    Отслеживание хода выполнения работ
    Пересмотр оценок параметров проекта
    Изменение графика работ
    Пересмотр проектных ограничений
    if (возникла проблема) then
        Пересмотр технических или организационных параметров проекта
    end if
end loop
```

Процесс планирования начинается с определения проектных ограничений (временные ограничения, возможности наличного персонала, бюджетные ограничения и т.д.). Эти ограничения должны определяться параллельно с оцениванием проектных параметров, таких как структура и размер проекта, а также распределением функций среди исполнителей. Затем определяются этапы разработки и то, какие результаты (документация, прототипы, подсистемы или версии программного продукта) должны быть получены по окончании этих этапов. Далее начинается циклическая часть планирования. Сначала разрабатывается график работ по выполнению проекта или дается разрешение на продолжение использования ранее созданного графика. После этого (обычно через 2–3 недели) проводится контроль выполнения работ и отмечаются расхождения между реальным и плановым ходом работ.

Далее, по мере поступления новой информации о ходе выполнения проекта, возможен пересмотр первоначальных оценок параметров проекта. Это, в свою очередь, может привести к изменению графика работ. Если в результате этих изменений нарушаются сроки завершения проекта, должны быть пересмотрены (и согласованы с заказчиком ПО) проектные ограничения.

Конечно, большинство менеджеров проектов не думают, что реализация их проектов пройдет гладко, без всяких проблем. Желательно описать возможные проблемы еще до того, как они проявят себя в ходе выполнения проекта. Поэтому лучше составлять "пессимистические" графики работ, чем "оптимистические". Но, конечно, невозможно построить план, учитывающий все, в том числе случайные, проблемы и задержки выполнения проекта, поэтому и возникает необходимость периодического пересмотра проектных ограничений и этапов создания программного продукта.

4.2.1. План проекта

План проекта должен четко показать ресурсы, необходимые для реализации проекта, разделение работ на этапы и временной график выполнения этих этапов. В некоторых организациях план проекта составляется как единый документ, содержащий все виды планов, описанных выше. В других случаях план проекта описывает только технологический процесс создания ПО. В таком плане обязательно присутствуют ссылки на планы других видов, но они разрабатываются отдельно от плана проекта.

План, структуру которого я представлю ниже, принадлежит именно к последнему типу планов. Детализация планов проектов очень разнится в зависимости от типа разрабатываемого программного продукта и организации-разработчика. Но в любом случае большинство планов содержат следующие разделы.

1. *Введение.* Краткое описание целей проекта и проектных ограничений (бюджетных, временных и т.д.), которые важны для управления проектом.

2. *Организация выполнения проекта.* Описание способа подбора команды разработчиков и распределение обязанностей между членами команды.
3. *Анализ рисков.* Описание возможных проектных рисков, вероятности их проявления и стратегий, направленных на их уменьшение. Тема управления рисками рассмотрена в разделе 4.4.
4. *Аппаратные и программные ресурсы, необходимые для реализации проекта.* Перечень аппаратных средств и программного обеспечения, необходимого для разработки программного продукта. Если аппаратные средства требуется закупать, приводится их стоимость совместно с графиком закупки и поставки.
5. *Разбиение работ на этапы.* Процесс реализации проекта разбивается на отдельные процессы, определяются этапы выполнения проекта, приводится описание результатов ("выходов") каждого этапа и контрольные отметки. Эта тема представлена в разделе 4.2.2.
6. *График работ.* В этом графике отображаются зависимости между отдельными процессами (этапами) разработки ПО, оценки времени их выполнения и распределение членов команды разработчиков по отдельным этапам.
7. *Механизмы мониторинга и контроля за ходом выполнения проекта.* Описываются предоставляемые менеджером отчеты о ходе выполнения работ, сроки их предоставления, а также механизмы мониторинга всего проекта.

План должен регулярно пересматриваться в процессе реализации проекта. Одни части плана, например график работ, изменяются часто, другие более стабильны. Для внесения изменений в план требуется специальная организация документопотока, позволяющая отслеживать эти изменения.

4.2.2. Контрольные отметки этапов работ

Менеджеру для организации процесса создания ПО и управления им необходима информация. Поскольку само программное обеспечение неосязаемо, эта управленческая информация может быть получена только в виде документов, отображающих выполнение очередного этапа разработки программного продукта. Без этой информации нельзя судить о степени готовности создаваемого продукта, невозможно оценить произведенные затраты или изменить график работ.

При планировании процесса определяются *контрольные отметки* — вехи, отмечающие окончание определенного этапа работ. Для каждой контрольной отметки создается отчет, который предоставляется руководству проекта. Эти отчеты не должны быть большими объемными документами; они должны подводить краткие итоги окончания отдельного логически завершенного этапа проекта. Этапом не может быть, например, "Написание 80% кода программ", поскольку невозможно проверить завершение такого "этапа"; кроме того, подобная информация практически бесполезна для управления, поскольку здесь не отображается связь этого "этапа" с другими этапами создания ПО.

Обычно по завершении основных больших этапов, таких как разработка спецификации, проектирование и т.п., заказчику ПО предоставляются результаты их выполнения, так называемые *контрольные проектные элементы.* Это может быть документация, прототип программного продукта, законченные подсистемы ПО и т.д. Контрольные проектные элементы, предоставляемые заказчику ПО, могут совпадать с контрольными отметками (точнее, с результатами выполнения какого-либо этапа). Но обратное утверждение неверно. Контрольные отметки — это внутренние проектные результаты, которые используются для контроля за ходом выполнения проекта, и они, как правило, не предоставляются заказчику ПО.

Для определения контрольных отметок весь процесс создания ПО должен быть разбит на отдельные этапы с указанным "выходом" (результатом) каждого этапа. Например, на рис. 4.1 показаны этапы разработки спецификации требований в случае, когда для ее проверки используется прототип системы, а также представлены выходные результаты (контрольные отметки) каждого этапа. Здесь контрольными проектными элементами являются требования и спецификация требований.

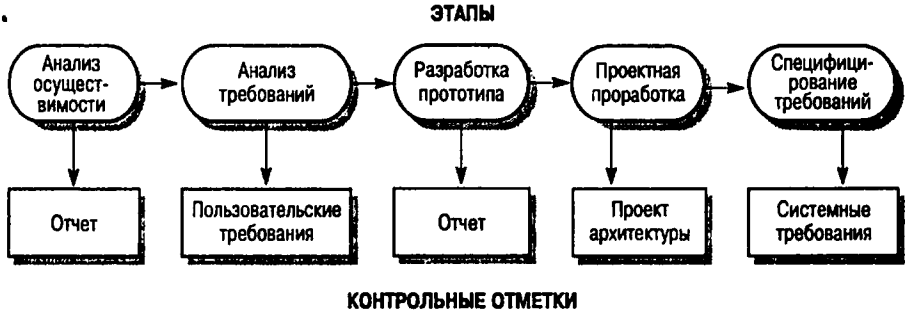


Рис. 4.1. Этапы процесса разработки спецификации

4.3. График работ

Составление графика – одна из самых ответственных работ, выполняемых менеджером проекта. Здесь менеджер оценивает длительность проекта, определяет ресурсы, необходимые для реализации отдельных этапов работ, и представляет их (этапы) в виде согласованной последовательности. Если данный проект подобен ранее реализованному, то график работ последнего проекта можно взять за основу для данного проекта. Но затем следует учесть, что на отдельных этапах нового проекта могут использоваться методы и подходы, отличные от использованных ранее.

Если проект является инновационным, первоначальные оценки длительности и требуемых ресурсов наверняка будут слишком оптимистичными, даже если менеджер попытается предусмотреть все возможные неожиданности. С этой точки зрения проекты создания ПО не отличаются от больших инновационных технических проектов. Новые аэропорты, мосты и даже новые автомобили, как правило, появляются позже первоначально объявленных сроков их сдачи или поступления на рынок, чему причиной являются неожиданно возникшие проблемы и трудности. Именно поэтому графики работ необходимо постоянно обновлять по мере поступления новой информации о ходе выполнения проекта.

В процессе составления графика (рис. 4.2) весь массив работ, необходимых для реализации проекта, разбивается на отдельные этапы и оценивается время, требующееся для выполнения каждого этапа. Обычно многие этапы выполняются параллельно. График работ должен предусматривать это и распределять производственные ресурсы между ними наиболее оптимальным образом. Нехватка ресурсов для выполнения какого-либо критического этапа – частая причина задержки выполнения всего проекта.

Длительность этапов обычно должна быть не меньше недели. Если она будет меньше, то окажется ниже точности временных оценок этапов, что может привести к частому пересмотру графика работ. Также целесообразно (в аспекте управления проектом) установить максимальную длительность этапов, не превышающую 8 или 10 недель. Если есть этапы, имеющие большую длительность, их следует разбить на этапы меньшей длительности.

При расчете длительности этапов менеджер должен учитывать, что выполнение любого этапа не обойдется без больших или маленьких проблем и задержек. Разработчики могут допускать ошибки или задерживать свою работу, техника может выйти из строя либо аппаратные или программные средства поддержки процесса разработки могут поступить с опозданием. Если проект инновационный и технически сложный, это становится дополнительным фактором появления непредвиденных проблем и увеличения длительности реализации проекта по сравнению с первоначальными оценками.

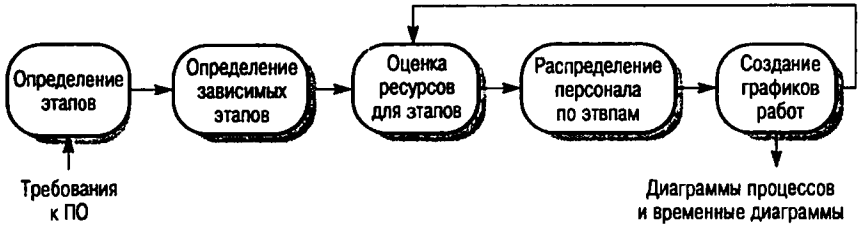


Рис. 4.2. Процесс составления графика работ

Кроме временных затрат, менеджер должен рассчитать другие ресурсы, необходимые для успешного выполнения каждого этапа. Особый вид ресурсов — это команда разработчиков, привлеченная к выполнению проекта. Другими видами ресурсов могут быть необходимое свободное дисковое пространство на сервере, время использования какого-либо специального оборудования и бюджетные средства на командировочные расходы персонала, работающего над проектом. Более детально виды и методы оценивания необходимых ресурсов рассматриваются в главе 23.

Существует хорошее эмпирическое правило: оценивать временные затраты так, как будто ничего непредвиденного и "плохого" не может случиться, затем увеличить эти оценки для учета возможных проблем. Возможные, но трудно прогнозируемые проблемы существенно зависят от типа и параметров проекта, а также от квалификации и опыта членов команды разработчиков. Так как это правило эмпирическое, позволю дать совет, основанный на моем опыте. К исходным расчетным оценкам я всегда добавляю 30% на возможные проблемы и затем еще 20%, чтобы быть готовым к тому, что я не могу предвидеть.

График работ по проекту обычно представляется в виде набора диаграмм и графиков, показывающих разбиение проектных работ на этапы, зависимости между этапами и распределение разработчиков по этапам. Эти диаграммы рассматриваются в следующем разделе. Отмечу, что в настоящее время существует много различных программных средств поддержки управления проектами, например Microsoft Project.

4.3.1. Временные и сетевые диаграммы

Временные и сетевые диаграммы полезны для представления графика работ. Временная диаграмма показывает время начала и окончания каждого этапа и его длительность. Сетевая диаграмма отображает зависимости между различными этапами проекта. Эти диаграммы можно создать автоматически с помощью программных средств поддержки управления на основе информации, заложенной в базе данных проекта.

Рассмотрим этапы некоего проекта, представленные в табл. 4.2, из которой, в частности, видно, что этап Т3 зависит от этапа Т1. Это значит, что этап Т1 должен завершиться прежде, чем начнется этап Т3. Например, на этапе Т1 проводится компонентный анализ создаваемого программного продукта, а на этапе Т3 — проектирование системы.

На основе приведенных значений длительности этапов и зависимости между ними строится сетевой график последовательности этапов (рис. 4.3). На этом графике видно, какие работы могут выполняться параллельно, а какие должны выполняться последовательно друг за другом. Этапы обозначены прямоугольниками. Контрольные отметки и контрольные проектные элементы показаны в виде овалов и обозначены (как и в табл. 4.2) буквой М с соответствующим номером. Даты на данной диаграмме соответствуют началу выполнения этапов. Сетевую диаграмму следует читать слева направо и сверху вниз.

Таблица 4.2. Этапы проекта

Этап	Длительность (дни)	Зависимость
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

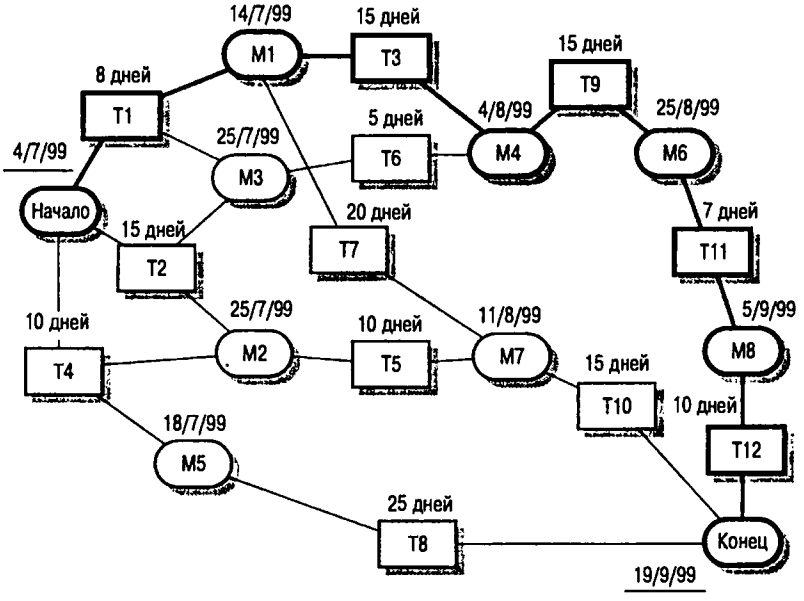


Рис. 4.3. Сетевая диаграмма этапов

Если для создания сетевой диаграммы используются программные средства поддержки управления проектом, каждый этап должен заканчиваться контрольной отметкой. Очередной этап может начаться только тогда, когда будет получена контрольная отметка (которая может зависеть от нескольких предшествующих этапов). Поэтому в третьем столбце табл. 4.2 приведены контрольные отметки; они будут достигнуты только тогда, когда будет завершен этап, в строке которого помещена соответствующая контрольная отметка.

Любой этап не может начаться, пока не выполнены все этапы на всех путях, ведущих от начала проекта к данному этапу. Например, этап Т9 не может начаться, пока не будут завершены этапы Т3 и Т6. Отметим, что в данном случае достижение контрольной отметки М4 говорит о том, что эти этапы завершены.

Минимальное время выполнения всего проекта можно рассчитать, просуммировав в сетевой диаграмме длительности этапов на самом длинном пути[1] от начала проекта до его окончания (это так называемый *критический путь*). В нашем случае продолжительность проекта составляет 11 недель или 55 рабочих дней. На рис. 4.3 критический путь показан более толстыми линиями, чем остальные пути. Таким образом, общая продолжительность реализации проекта зависит от этапов работ, находящихся на критическом пути. Любая задержка в завершении любого этапа на критическом пути приведет к задержке всего проекта.

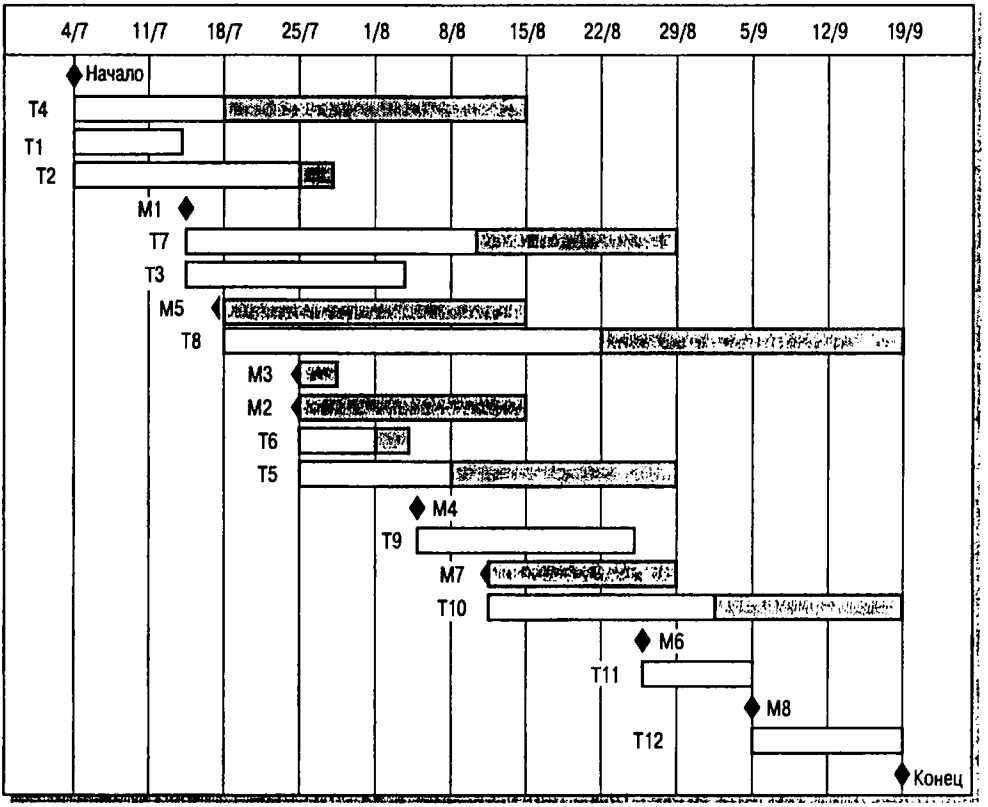


Рис. 4.4. Временная диаграмма длительности этапов

[1] *Длина пути здесь измеряется не количеством этапов на пути, а суммарной длительностью этих этапов.* – Прим. ред.

Задержка в завершении этапов, не входящих в критический путь, не влияет на продолжительность всего проекта до тех пор, пока суммарная длительность этих этапов (с учетом задержек) на каком-нибудь пути не превысит продолжительности работ на критическом пути. Например, задержка этапа Т8 на срок, меньший 20 дней, никак не влияет на общую продолжительность проекта. На рис. 4.4 представлена временная диаграмма, на которой показаны возможные задержки на каждом этапе.

Сетевая диаграмма позволяет увидеть в зависимости этапов значимость того или иного этапа для реализации всего проекта. Внимание к этапам критического пути часто позволяет найти способы их изменения с тем, чтобы сократить длительность всего проекта. Менеджеры используют сетевую диаграмму для распределения работ.

На рис. 4.4 показано другое представление графика работ. Это временная диаграмма (иногда называемая по имени ее изобретателя диаграммой Гантта) может быть построена программными средствами поддержки процесса управления. Она показывает длительность выполнения каждого этапа и возможные их задержки (показаны затененными прямоугольниками), а также даты начала и окончания каждого этапа. Этапы критического пути не имеют затененных прямоугольников; это означает, что задержка с завершением данных этапов приведет к увеличению длительности всего проекта.

Подобно распределению времени выполнения этапов, менеджер должен рассчитать распределение ресурсов по этапам, в частности назначить исполнителей на каждый этап. В табл. 4.3 приведено распределение разработчиков на каждый этап, представленный на рис. 4.4.

Таблица 4.3. Распределение исполнителей по этапам

Этап	Исполнитель
Т1	Джейн
Т2	Анна
Т3	Джейн
Т4	Фред
Т5	Мэри
Т6	Анна
Т7	Джим
Т8	Фред
Т9	Джейн
Т10	Анна
Т11	Фред
Т12	Фред

Приведенная таблица может быть использована программными средствами поддержки процесса управления для построения временной диаграммы занятости сотрудников на определенных этапах работ (рис. 4.5). Персонал не занят в работе над проектом все время его реализации. В течение периода незанятости сотрудники могут быть в отпуске, работать над другими проектами, проходить обучение и т.д.

В больших организациях обычно работает много специалистов, которые задействуются в проекте по мере необходимости. Конечно, такой подход может создать определенные проблемы для менеджеров проектов. Например, если специалист занят в проекте, который задерживается, это может создать прямые сложности для других проектов, где он также должен участвовать.

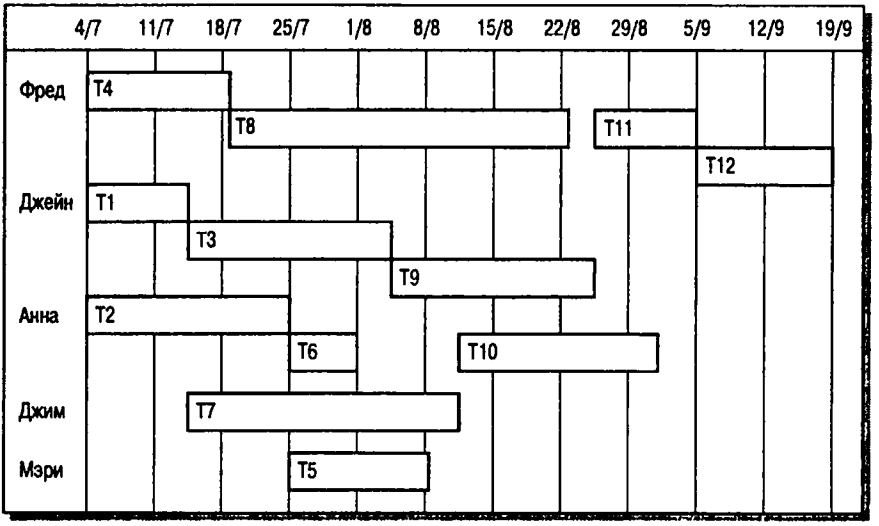


Рис. 4.5. Временная диаграмма распределения работников по этапам

Первоначальный график работ неизбежно содержит какие-нибудь ошибки или недоработки. По мере реализации проекта рассчитанные оценки длительности выполнения этапов работ должны сравниваться с реальными сроками выполнения этих этапов. Результаты сравнения должны использоваться в качестве основы для пересмотра графика работ еще не реализованных этапов проекта, в частности для того, чтобы попытаться уменьшить длительность этапов критического пути.

4.4. Управление рисками

Важной частью работы менеджера проекта является оценка рисков, которые могут повлиять на график работ или на качество создаваемого программного продукта, и разработка мероприятий по предотвращению рисков. Результаты анализа рисков должны быть отражены в плане проекта. Определение рисков и разработка мероприятий по уменьшению их влияния на ход выполнения проекта называется *управлением рисками* [149, 266, 11*, 23*].

Упрощенно риск можно понимать как вероятность проявления каких-либо неблагоприятных обстоятельств, негативно влияющих на реализацию проекта. Риски могут угрожать проекту в целом, создаваемому программному продукту или организации-разработчику. Можно выделить три типа рисков.

1. *Риски для проекта*, которые влияют на график работ или ресурсы, необходимые для выполнения проекта.

2. *Риски для разрабатываемого продукта*, влияющие на качество или производительность разрабатываемого программного продукта.
3. *Бизнес-риски*, относящиеся к организации-разработчику или поставщикам.

Конечно, эти типы рисков могут пересекаться. Например, если опытный программист покидает проект, это будет риском для проекта (поскольку задерживается срок сдачи готового продукта), риском для продукта (так как новый программист, заменивший ушедшего, может оказаться не слишком опытным и сделать ошибки в программе) и бизнес-риском (поскольку задержка данного проекта может негативно повлиять на будущие деловые контакты между заказчиком и организацией-разработчиком).

Конкретные типы рисков, которые могут оказать влияние на данный проект, зависят от вида создаваемого программного продукта и от организационного окружения, где реализуется программный проект. Вместе с тем многие типы рисков способны повлиять на любые программные проекты, эти риски приведены в табл. 4.4.

Таблица 4.4. Возможные риски программных проектов

Риск	Тип риска	Описание риска
Текучесть разработчиков	Риск для проекта	Опытные разработчики покидают проект до его завершения
Изменение в управлении организацией	Риск для проекта	Организация меняет свои приоритеты в управлении проектом
Неготовность аппаратных средств	Риск для проекта	Аппаратные средства, которые необходимы для проекта, не поступили вовремя или не готовы к эксплуатации
Изменение требований	Риск для проекта и для разрабатываемого продукта	Появление большого количества непредвиденных изменений в требованиях, предъявляемых к разрабатываемому ПО
Задержка в разработке спецификации	Риск для проекта и для разрабатываемого продукта	Спецификации основных интерфейсов подсистем не поступили к разработчикам в соответствии с графиком работ
Недооценка размера разрабатываемой системы	Риск для проекта и для разрабатываемого продукта	Размер системы значительно превысил первоначальную оценку
Недостаточная эффективность CASE-средств	Риск для разрабатываемого продукта	CASE-средства, предназначенные для поддержки проекта, оказались менее эффективными, чем ожидалось
Изменения в технологии разработки ПО	Бизнес-риск	Основные технологии построения программной системы заменяются новыми
Появление конкурирующего программного продукта	Бизнес-риск	На рынке программных продуктов до окончания проекта появилась конкурирующая программная система

Схема процесса управления рисками показана на рис. 4.6. Этот процесс состоит из четырех стадий.

1. *Определение рисков.* Определяются возможные риски для проекта, для разрабатываемого продукта и бизнес-риски.
2. *Анализ рисков.* Оценивается вероятность и последовательность появления рисковых ситуаций.
3. *Планирование рисков.* Планируются мероприятия по предотвращению рисков или минимизации их воздействия на проект.
4. *Мониторинг рисков.* Постоянное оценивание вероятностей рисков и выполнение мероприятий по смягчению последствий проявления рисковых ситуаций.

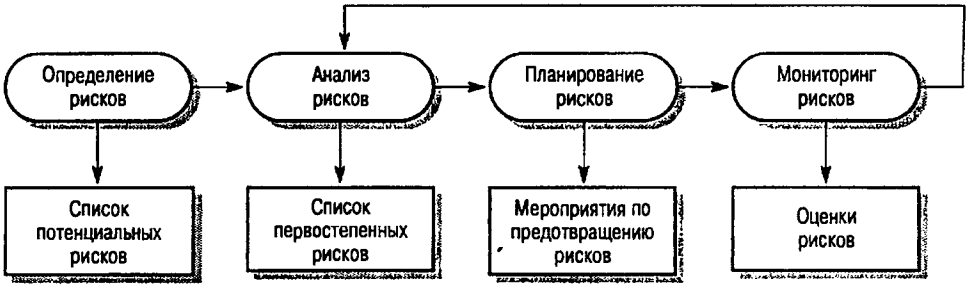


Рис. 4.6. Процесс управления рисками

Процесс управления рисками, как и другие процессы планирования, является итерационным, выполняемым в течение всего срока реализации проекта. Сначала разрабатываются планы управления рисками, затем постоянно отслеживается ситуация вокруг реализации проекта. При поступлении новой информации о возможных рисках заново проводится анализ рисков и первостепенное внимание уделяется новым рискам. По мере поступления новой информации также изменяются планы мероприятий по предотвращению и смягчению рисков.

Результаты процесса управления рисками документируются в виде планов управления рисками. Они должны включать описание возможных проектных рисков, их анализ и перечень мероприятий, необходимых для управления рисками.

4.4.1. Определение рисков

Определение рисков – первая стадия процесса управления рисками. На этой стадии описываются риски, которые могут проявиться при реализации проекта. В принципе на этой стадии не должна оцениваться вероятность и значимость рисков, но на практике маловероятные риски с незначительными последствиями обычно отбрасываются сразу.

Определение рисков может выполняться в режиме командной работы с использованием подхода "мозговой штурм" либо основываться на опыте менеджера. При определении рисков может помочь приведенный ниже список возможных категорий рисков.

1. *Технологические риски.* Проистекают из программных и аппаратных технологий, на основе которых разрабатывается система.
2. *Риски, связанные с персоналом.* Связаны с членами команды разработчиков.

3. *Организационные риски.* Проистекают из организационного окружения, в котором выполняется проект.
4. *Инструментальные риски.* Связаны с используемыми CASE-средствами и другими средствами поддержки процесса создания ПО.
5. *Риски, связанные с системными требованиями.* Проявляются при изменении требований, предъявляемых к разрабатываемой системе.
6. *Риски оценивания.* Связаны с оцениванием характеристик программной системы и ресурсов, необходимых для реализации проекта.

В табл. 4.5 представлены некоторые примеры, относящиеся к каждой из описанных категорий рисков. Результатом этапа определения рисков будет длинный список возможных рисков, которые могут повлиять на разрабатываемый программный продукт, проект или организацию-разработчика.

Таблица 4.5. Категории рисков

Категория рисков	Примеры рисков
Технологические риски	База данных, которая используется в программной системе, не обеспечивает обработку ожидаемого объема транзакций.
	Программные компоненты, которые используются повторно, имеют дефекты, ограничивающие их функциональные возможности
Риски, связанные с персоналом	Невозможно подобрать работников с требуемым профессиональным уровнем.
	Ведущий разработчик заболел в самое критическое время.
	Невозможно организовать необходимое обучение персонала
Организационные риски	В организации, выполняющей разработку ПО, произошла реорганизация, в результате чего изменились приоритеты в управлении проектом.
	Финансовые затруднения в организации привели к уменьшению бюджета проекта
Инструментальные риски	Программный код, генерируемый CASE-средствами, не эффективен.
	CASE-средства невозможно интегрировать с другими средствами поддержки проекта
Риски, связанные с системными требованиями	Изменения требований приводят к значительным повторным работам по проектированию системы.
	Первоначальная нечеткая формулировка пользовательских требований привела к значительным изменениям системных требований, проявившихся на поздних стадиях разработки проекта

Окончание табл. 4.5

Категория рисков	Примеры рисков
Риски оценивания	Недооценки времени выполнения проекта.
	Скорость выявления дефектов в системе ниже ранее запланированной.
	Размер системы значительно превышает первоначально рассчитанный

4.4.2. Анализ рисков

При анализе для каждого определенного риска подсчитывается вероятность его проявления и ущерб, который он может нанести. Не существует простых методов выполнения анализа рисков — в значительной мере он основан на мнении и опыте менеджера. Не претендуя на исключительную точность, можно привести следующую шкалу вероятностей рисков и их последствий.

1. Вероятность риска считается очень низкой, если она имеет значение менее 10%; низкой, если ее значение от 10 до 25 %; средней при значениях от 25 до 50%; высокой, если значение колеблется от 50 до 75%; очень высокой при значениях более 75%.
2. Возможный ущерб от рисковых ситуаций можно подразделить на катастрофический, серьезный, терпимый и незначительный.

Результаты анализа рисков должны быть представлены в виде таблицы рисков, упорядоченных по степени возможного ущерба. В табл. 4.6 приведен упорядоченный список рисков, описанных в табл4.5; там же указаны вероятности этих рисков. Здесь вероятности рисков и степень ущерба от них указаны произвольно. На практике для их определения необходима подробная информация о проекте, технологии создания ПО, команде разработчиков и о самой организации.

Таблица 4.6. Список рисков после проведения их анализа

Риск	Вероятность	Степень ущерба
Финансовые затруднения в организации привели к уменьшению бюджета проекта	Низкая	Катастрофическая
Невозможно подобрать работников с требующимся профессиональным уровнем	Высокая	Катастрофическая
Ведущий разработчик заболел в самое критическое время	Средняя	Серьезная
Программные компоненты, используемые повторно, имеют дефекты, ограничивающие их функциональные возможности	Средняя	Серьезная
Изменения требований приводят к значительным повторным работам по проектированию системы	Средняя	Серьезная
В организации, выполняющей разработку ПО, произошла реорганизация, в результате чего изменились приоритеты в управлении проектом	Высокая	Серьезная

Окончание табл. 4.6

Риск	Вероятность	Степень ущерба
База данных, которая используется в программной системе, не обеспечивает обработку ожидаемого объема транзакций	Средняя	Серьезная
Недооценки времени выполнения проекта	Высокая	Серьезная
CASE-средства невозможно интегрировать с другими средствами поддержки проекта	Высокая	Терпимая
Первоначальная нечеткая формулировка пользовательских требований привела к значительным изменениям системных требований, проявившихся на поздних стадиях разработки проекта	Средняя	Терпимая
Невозможно организовать необходимое обучение персонала	Средняя	Терпимая
Скорость выявления дефектов в системе ниже ранее запланированной	Средняя	Терпимая
Размер системы значительно превышает первоначально рассчитанный	Высокая	Терпимая
Программный код, генерируемый CASE-средствами, неэффективен	Средняя	Незначительная

Конечно, как вероятность рисков, так и возможный ущерб от них должны пересматриваться при поступлении дополнительной информации об этих рисках и по мере реализации мероприятий по управлению ими. Поэтому подобные таблицы рисков должны переделываться на каждой итерации процесса управления рисками.

После проведения анализа рисков определяются наиболее значимые риски, которые затем отслеживаются на протяжении всего срока выполнения проекта. Определение этих значимых рисков зависит от их вероятностей и возможного ущерба. В общем случае всегда отслеживаются риски с катастрофическими последствиями, а также риски с серьезным ущербом, значение вероятности которых выше среднего.

В статье [47] рекомендуется определить и отслеживать "10 верхних" рисков, но я думаю, что это необоснованная рекомендация. Количество рисков, которые необходимо отслеживать, зависит от конкретного проекта. Это может быть пять рисков, а может — пятнадцать. Но, конечно, количество рисков, по которым проводится мониторинг, должно быть обозримым. Большое количество отслеживаемых рисков потребует огромного количества собираемой информации. Из списка рисков, представленных в табл. 4.6, для мониторинга следует отобрать все восемь рисков, которые могут привести к катастрофическим и серьезным последствиям.

4.4.3. Планирование рисков

Планирование заключается в определении стратегии управления каждым значимым риском, отобранным для мониторинга после анализа рисков. Здесь также не существует общепринятых подходов для разработки таких стратегий — многое основывается на "чутье" и опыте менеджера проекта. В табл. 4.7 показаны возможные стратегии управления основными рисками, приведенными в табл. 4.6.

Таблица 4.7. Стратегии управления рисками

Риск	Стратегия
Финансовые проблемы организации	Подготовить краткий документ для руководства организации, показывающий важность данного проекта для достижения финансовых целей организации
Проблемы неквалифицированного персонала	Предупреднть заказчика о потенциальных трудностях и возможной задержке проекта, рассмотреть вопрос о покупке компонентов системы
Болезни персонала	Реорганизовать работу команды разработчиков таким образом, чтобы обязанности и работа членов команды перекрывали друг друга, вследствие этого разработчики будут знать и понимать задачи, выполняемые другими сотрудниками
Дефектные системные компоненты	Заменить потенциально дефектные системные компоненты покупными компонентами, гарантирующими качество работы
Изменения требований	Попытаться определить требования, наиболее вероятно подверженные изменениям; в структуре системы не отображать детальную информацию
Реорганизация компании-разработчика	Подготовить краткий документ для руководства компании, показывающий важность данного проекта для достижения финансовых целей компании
Недостаточная производительность базы данных	Рассмотреть возможность покупки более производительной базы данных
Недооценки времени выполнения проекта	Рассмотреть вопрос о покупке системных компонентов, исследовать возможность использования генератора программного кода

Существует три категории стратегий управления рисками.

1. *Стратегии предотвращения рисков.* Согласно этим стратегиям следует проводить мероприятия, снижающие вероятность проявления рисков. Примером может служить стратегия исключения потенциально дефектных компонентов, описанная в табл. 4.7.
2. *Минимизационные стратегии.* Направлены на уменьшение возможного ущерба от рисков. Примером служит стратегия уменьшения ущерба от болезни членов команды разработчиков (см. табл. 4.7).
3. *Планирование "аварийных" ситуаций.* Согласно этим стратегиям необходимо иметь план мероприятий, которые следует выполнить в случае проявления рисковой ситуации. В табл. 4.7 это стратегия поведения при возникновении финансовых проблем у организации-разработчика.

4.4.4. Мониторинг рисков

Мониторинг рисков заключается в регулярном пересчете вероятностей рисков и ущерба, который они могут нанести. Для этого необходимо постоянно отслеживать факторы, которые влияют на вероятность рисков и возможный ущерб. Эти факторы зависят от типов риска. В табл. 4.8 приведены признаки, которые помогают определить тип риска.

Таблица 4.8. Признаки рисков

Тип риска	Признаки
Технологические риски	Задержки в поставке оборудования или программных средств поддержки процесса создания ПО, многочисленные документированные технологические проблемы
Риски, связанные с персоналом	Низкое моральное состояние персонала, натянутые отношения между членами команды разработчиков, низкое качество выполненной работы
Организационные риски	Разговоры среди персонала о пассивности и недостаточной компетентности высшего руководства организации
Инструментальные риски	Нежелание разработчиков использовать программные средства поддержки, неодобрительные отзывы о CASE-средствах, запросы на более мощные инструментальные средства
Риски, связанные с системными требованиями	Необходимость пересмотра многих системных требований, недовольство заказчика ПО
Риски оценивания	Изменения графика работ, многочисленные отчеты о нарушении графика работ

Мониторинг рисков должен быть непрерывным процессом, отслеживающим ход выполнения мероприятий по управлению рисками, при этом каждый основной риск должен рассматриваться отдельно.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Признаком хорошего управления программным проектом является выполнение проекта в соответствии с графиком работ и в рамках запланированного бюджета.
- Управление программными проектами отличается от управления другими техническими проектами, поскольку программное обеспечение нематериально. Опыта, накопленного при реализации более ранних проектов, может оказаться недостаточно для выработки стратегии управления новыми и сложными программными проектами.
- Менеджер программного проекта выполняет множество разнотипных работ. Среди них основными являются планирование проекта, оценивание временных и иных ресурсов, необходимых для реализации проекта, а также составление графика работ. Планирование проекта и оценивание необходимых ресурсов — это итерационные виды деятельности, которые продолжаются в течение всего срока выполнения проекта. По мере поступления дополнительной информации о ходе выполнения проекта планы и графики могут пересматриваться.
- Контрольные отметки — это прогнозируемые "выходы" этапов реализации проектов, которые совместно с отчетом о выполнении этапа передаются руководству проектом. Контрольные проектные элементы — это контрольные отметки, которые предоставляются заказчику программной системы.
- Составление графика работ состоит в создании различных графических представлений отдельных частей плана проекта. Сюда относятся сетевые диаграммы этапов, показывающие взаимозависимость этапов работ, и временные диаграммы длительности этапов.
- Для любого проекта должны быть определены основные риски, а также вероятность их осуществления и возможный ущерб от них. Для наиболее вероятных рисков, последствия от которых более чем серьезны, разрабатываются мероприятия по их предотвращению или снижению возможного ущерба.

Упражнения

4.1. Объясните, почему нематериальность программных систем порождает особые проблемы в процессе управления программными проектами.

4.2. Объясните, почему хорошие программисты не всегда могут быть хорошими менеджера проектов. Для построения объяснения может помочь список работ, выполняемых менеджером проектов (см. раздел 4.1).

4.3. Объясните, почему процесс планирования проекта является итерационным и почему план должен постоянно пересматриваться в течение всего срока выполнения проекта.

4.4. Опишите кратко каждый раздел плана выполнения программного проекта.

4.5. В чем принципиальное различие между контрольной отметкой и контрольным программным элементом?

4.6. В табл. 4.9 показаны этапы некоего проекта, их длительность и взаимозависимость между ними. Нарисуйте сетевую и временную диаграммы этапов работ для данного проекта.

Таблица 4.9. Этапы проекта

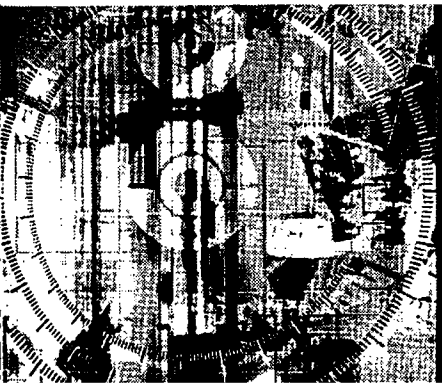
Этап	Длительность (дни)	Зависимость
T1	10	
T2	15	T1
T3	10	T1, T2
T4	20	
T5	10	
T6	15	T3, T4
T7	20	T3
T8	35	T7
T9	15	T6
T10	5	T5, T9
T11	10	T9
T12	20	T10
T13	35	T3, T4
T14	10	T8, T9
T15	20	T12, T14
T16	10	T15

4.7. В табл. 4.2 приведена длительность этапов некоторого проекта. Предположим, что вследствие неких серьезных причин этап T5 был выполнен за 40 дней вместо запланированных 10 дней. Переделайте сетевую диаграмму этапов и определите новый критический путь. Нарисуйте новую временную диаграмму.

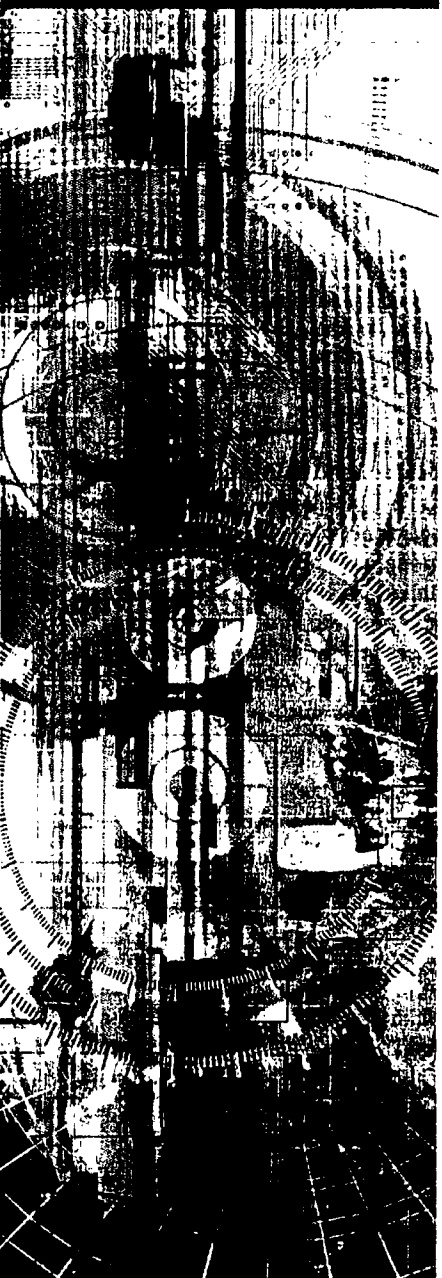
4.8. В дополнение к рискам, приведенным в табл. 4.5, определите еще шесть рисков, которые возможны во время реализации проекта.

4.9. Менеджер проекта предупреждает о возможной задержке выполнения работ, которой можно избежать только за счет бесплатных сверхурочных работ команды разработчиков. Все члены команды имеют семьи, требующие определенной доли внимания. Обсудите возможность отклонения предложения менеджера о бесплатных сверхурочных работах либо согласия предпочесть интересы организации семейным интересам. Какие аргументы наиболее весомы в этой дискуссии?

4.10. Как опытному программисту, вам предложили возглавить управление проектом, но вы чувствуете, что больше пользы можете принести в качестве технического специалиста, а не менеджера проекта. Обсудите возможности принятия или отклонения предложения возглавить программный проект.



Требования



5

Требования к программному обеспечению

Цели

Цель настоящей главы — дать основные понятия
требованиях, предъявляемых к программным сис
мам, и показать различные способы представлен
этих требований. Прочитав эту главу, вы должны:

- иметь понятие о концепциях пользовательских
системных требований и знать, почему для запи
этих требований используются разные способы;
- понимать различия между функциональными
нефункциональными требованиями;
- освоить два метода описания системных треб
ваний: основанный на структурированном ест
ственном языке и основанный на языке пр
граммирования;
- знать стандарты документирования требований
программному обеспечению.

Содержание

Проблемы, которые приходится решать специалистам в процессе создания программного обеспечения, обычно очень сложны. Природа этих проблем не всегда ясна, особенно если разрабатываемая программная система инновационная. В частности, трудно четко описать те действия, которые должна выполнять система. Описание функциональных возможностей и ограничений, накладываемых на программную систему, называется *требованиями* к этой системе, а сам процесс формирования, анализа, документирования и проверки этих функциональных возможностей и ограничений — *разработкой требований* (requirements engineering). В этой главе внимание концентрируется на самих требованиях и способах их описания. Процесс разработки требований в общих чертах описан в главе 3, а более подробно освещен в следующей главе.

Термин *требования* (к программной системе) может трактоваться по-разному. В некоторых случаях под требованиями понимаются высокоуровневые обобщенные утверждения о функциональных возможностях и ограничениях системы. Другая крайняя ситуация — детализированное математическое формальное описание системных функций. Дэвис (Davis) [89] так объясняет причины этих различий.

> Если компания хочет выиграть контракт на разработку большого программного проекта, она вынуждена, пока решение не принято, представлять требования в самом обобщенном виде, чтобы, с одной стороны, удовлетворить требования заказчика, а с другой — иметь возможность для маневра при конкуренции с другими компаниями-разработчиками. После того как контракт выигран, компания должна представить заказчику более подробное описание системы с указанием всех выполняемых ею функций. В обеих ситуациях предоставляются документы, которые называются *документированными требованиями* к системе.

Некоторые проблемы, возникающие в процессе разработки требований, порождены отсутствием четкого понимания различия между этими разными уровнями требований. Чтобы различить требования разных уровней, здесь используются термины *пользовательские требования* (user requirements) для обозначения высокоуровневых обобщенных требований и *системные требования* (system requirements) для детализированного описания выполняемых системой функций. Кроме требований этих двух уровней, применяется еще более детализированное описание системы — *проектная системная спецификация* (software design specification), которая может служить мостом между этапом разработки требований и этапом проектирования системы. Три перечисленных вида требований можно определить следующим образом.

1. *Пользовательские требования* — описание на естественном языке (плюс поясняющие диаграммы) функций, выполняемых системой, и ограничений, накладываемых на нее.
2. *Системные требования* — детализированное описание системных функций и ограничений, которое иногда называют функциональной спецификацией. Она служит основой для заключения контракта между покупателем системы и разработчиками ПО.
3. *Проектная системная спецификация* — обобщенное описание структуры программной системы, которое будет основой для более детализированного проектирования системы и ее последующей реализации. Эта спецификация дополняет и детализирует спецификацию системных требований.

Различие между пользовательскими и системными требованиями показаны в примере, представленном в табл. 5.1. Здесь показано, как пользовательские требования могут быть преобразованы в системные.

Таблица 5.1. Пользовательские и системные требования

Пользовательские требования
1. ПО должно предоставить средство доступа к внешним файлам, созданным в других программах.
Спецификация системных требований
1.1. Пользователь должен иметь возможность определять тип внешних файлов.
1.2. Для каждого типа внешнего файла должно иметься соответствующее средство, применимое к этому типу файлов.
1.3. Внешний файл каждого типа должен быть представлен соответствующей пиктограммой на дисплее пользователя.
1.4. Пользователю должна быть предоставлена возможность самому определять пиктограмму для каждого типа внешних файлов.
1.5. При выборе пользователем пиктограммы, представляющей внешний файл, к этому файлу должно быть применено средство, ассоциированное с внешними файлами данного типа.

Пользовательские требования пишутся для заказчика ПО и для лица, заключающего контракт на разработку программной системы, причем они могут не иметь детальных технических знаний по разрабатываемой системе (рис. 5.1). Спецификация системных требований предназначена для руководящего технического состава компании-разработчика и для менеджеров проекта. Она также необходима заказчику ПО и субподрядчикам по разработке. Эти оба документа также предназначены для конечных пользователей программной системы. Наконец, проектная системная спецификация является документом, который ориентирован на разработчиков ПО.

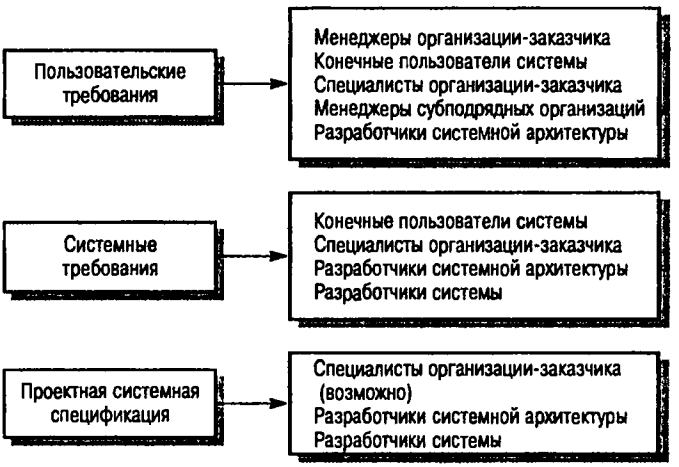


Рис. 5.1. Различные типы спецификаций требований и их читатели

5.1. Функциональные и нефункциональные требования

Требования к программной системе часто классифицируются как функциональные, нефункциональные и требования предметной области.

1. *Функциональные требования.* Это перечень сервисов, которые должна выполнять система, причем должно быть указано, как система реагирует на те или иные входные данные, как она ведет себя в определенных ситуациях и т.д. В некоторых случаях указывается, что система не должна делать.
2. *Нефункциональные требования.* Описывают характеристики системы и ее окружения, а не поведение системы. Здесь также может быть приведен перечень ограничений, накладываемых на действия и функции, выполняемые системой. Они включают временные ограничения, ограничения на процесс разработки системы, стандарты и т.д.
3. *Требования предметной области.* Характеризуют ту предметную область, где будет эксплуатироваться система. Эти требования могут быть функциональными и нефункциональными.

В действительности четкой границы между этими типами требований не существует. Например, пользовательские требования, касающиеся безопасности системы, можно отнести к нефункциональным. Однако при более детальном рассмотрении такое требование можно отнести к функциональным, поскольку оно порождает необходимость включения в систему средства авторизации пользователя. Поэтому, рассматривая далее эти виды требований, мы должны всегда помнить, что данная классификация в значительной степени искусственна.

5.1.1. Функциональные требования

Эти требования описывают поведение системы и сервисы (функции), которые она выполняет, и зависят от типа разрабатываемой системы и от потребностей пользователей. Если функциональные требования оформлены как пользовательские, они, как правило, описывают системы в обобщенном виде. В противоположность этому функциональные требования, оформленные как системные, описывают систему максимально подробно, включая ее входные и выходные данные, исключения и т.д.

Функциональные требования для программных систем могут быть описаны разными способами. Рассмотрим для примера функциоиальные требования к библиотечной системе университета, предназначенной для заказа книг и документов из других библиотек [204].

1. Пользователь должен иметь возможность проводить поиск необходимых ему книг и документов или по всему множеству доступных каталожных баз данных или по определенному их подмножеству.
2. Система должна предоставлять пользователю подходящее средство просмотра библиотечных документов.
3. Каждый заказ должен быть снабжен уникальным идентификатором (ORDER_ID), который копируется в формуляр пользователя для постоянного хранения.

Эти функциональные пользовательские требования определяют свойства, которыми должна обладать система. Они взяты из документа, содержащего пользовательские требования, и показывают, что функциональные требования могут быть описаны с разным уровнем детализации (сравните первое и третье требования).

Многие проблемы, возникающие при разработке систем, связаны с неточностью и "размытостью" спецификации требований. Естественно, разработчики интерпретируют требования, допускающие двоякое толкование, так, чтобы систему было проще реализовать. Но это толкование может не совпадать с ожиданиями заказчика. Такая ситуация приводит к разработке новых требований и внесению изменений в систему. Это, в свою очередь, ведет к задержке сдачи готовой системы и ее удорожанию.

Рассмотрим второе требование к библиотечной системе из приведенного выше списка и обратим внимание на выражение "подходящее средство просмотра документов". Библиотечная система может предоставлять документы в широком спектре форматов. В требовании подразумевается, что система должна предоставить средства для просмотра документов в любом формате. Но поскольку это условие четко не выписано, разработчики в случае дефицита времени могут использовать простое средство для просмотра текстовых документов и настаивать на том, что именно такое решение следует из данного требования.

В принципе спецификация функциональных требований должна быть комплексной и непротиворечивой. Комплексность подразумевает описание (определение) всех системных сервисов. Непротиворечивость означает отсутствие несовместимых и взаимоисключающих определений сервисов. На практике для больших и сложных систем крайне трудно разработать комплексную и непротиворечивую спецификацию функциональных требований. Причина кроется частично в сложности самой разрабатываемой системы, а частично – в несогласованных опорных точках зрения (см. главу 6) на то, что должна делать система. Эта несогласованность может не проявиться на этапе первоначального формулирования требований – для ее выявления необходим более глубокий анализ спецификации. Когда несогласованность системных функций проявится на каком-либо этапе жизненного цикла программы, в системную спецификацию придется внести соответствующие изменения.

5.1.2. Нефункциональные требования

Как следует из названия, нефункциональные требования не связаны непосредственно с функциями, выполняемыми системой. Они связаны с такими интегрированными свойствами системы, как надежность, время ответа или размер системы. Кроме того, нефункциональные требования могут определять ограничения на систему, например на пропускную способность устройств ввода-вывода, или форматы данных, используемых в системном интерфейсе.

Многие нефункциональные требования относятся к системе в целом, а не к отдельным ее средствам. Это означает, что они более значимы и критичны, чем отдельные функциональные требования. Ошибка, допущенная в функциональном требовании, может снизить качество системы, ошибка в нефункциональных требованиях может сделать систему неработоспособной.

Вместе с тем нефункциональные требования могут относиться не только к самой программной системе: одни могут относиться к технологическому процессу создания ПО, другие – содержать перечень стандартов качества, накладываемых на процесс разработки. Кроме того, в спецификации нефункциональных требований может быть указано, что проектирование системы должно выполняться только определенными CASE-средствами, и приведено описание процесса проектирования, которому необходимо следовать.

Нефункциональные требования отображают пользовательские потребности; при этом они основываются на бюджетных ограничениях, учитывают организационные возможности компании-разработчика и возможность взаимодействия разрабатываемой системы с другими программными и вычислительными системами, а также такие внешние факторы, как правила техники безопасности, законодательство о защите интеллектуальной собственности и т.п. На рис. 5.2 показана классификация нефункциональных требований.

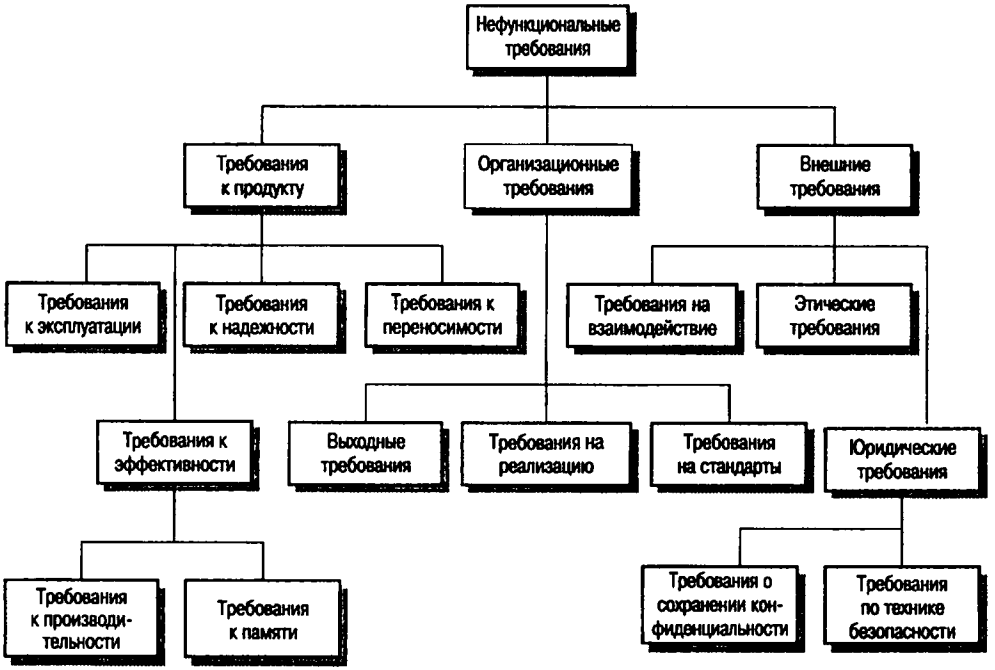


Рис. 5.2. Типы нефункциональных требований

Все нефункциональные требования, показанные на рис. 5.2, я разбил на три большие группы.

1. *Требования к продукту.* Описывают эксплуатационные свойства программного продукта. Сюда относятся требования к производительности системы, объему необходимой памяти, надежности (определяет частоту возможных сбоев в системе), переносимости системы на разные компьютерные платформы и удобству эксплуатации.
2. *Организационные требования.* Отображают политику и организационные процедуры заказчика и разработчика ПО. Они включают стандарты разработки программного продукта, требования к реализации ПО (т.е. к языку программирования и методам проектирования), выходные требования, которые определяют сроки изготовления программного продукта, и сопутствующую документацию.
3. *Внешние требования.* Учитывают факторы, внешние по отношению к разрабатываемой системе и процессу ее разработки. Они включают требования, определяющие взаимодействие данной системы с другими системами, юридические требования, следование которым гарантирует, что система будет разрабатываться и функционировать в рамках существующего законодательства, а также этические требования. Последние должны гарантировать, что система будет приемлемой для пользователей или заказчика.

Во врезке 5.1 приведен пример требований к продукту, организационных и внешних требований. Требования к продукту связаны со средой программирования APSE для языка Ada. Это ограничивает свободу проектировщика системы в выборе символов — можно использовать только символы из пользовательского интерфейса APSE. Организационные требования указывают, что система должна разрабатываться согласно внутреннему стандарту компании на разработку ПО, имеющему код XYZCo-SP-STAN-95. Внешние требова-

ния вытекают из необходимости соблюдения законодательства о сохранении конфиденциальности. Вследствие этого системные операторы не будут иметь доступ к тем данным, которые им не требуются для работы с системой.

Врезка 5.1. Пример нефункциональных требований

Требования к продукту

4.С.8. Все взаимодействия между интерфейсом APSE и пользователем осуществляются на основе стандартного множества символов языка Ada.

Организационные требования

Разработка системы и создание сопутствующей документации выполняются на основе стандарта XYZCo-SP-STAN-95.

Внешние требования

7.6.5. Система не должна раскрывать конфиденциальной информации о заказчике системы, кроме его имени, а также телефонного номера системных операторов.

Основная проблема нефункциональных требований состоит в том, что их выполнение трудно проверить. Часто они пишутся для того, чтобы отобразить общие цели заказчика системы, такие, как простота эксплуатации, возможность восстановления после сбоев или быстрый ответ на запросы пользователя. Реализация подобных требований может оказаться сложной для системных разработчиков, поскольку они нечетко сформулированы и открывают простор для различных толкований. Подобную ситуацию иллюстрирует пример, приведенный во врезке 5.2. Здесь одним из основных показателей (целей) системы указана простота эксплуатации, что в виде нефункциональных требований можно выразить различными способами. В данном случае требование сформулировано так, что его можно проверить.

Врезка 5.2. Системные цели и проверка требований

Системная цель

Система должна быть простой в эксплуатации для опытного оператора и сводить количество его ошибок к минимуму.

Проверяемое нефункциональное требование

Опытному оператору должны быть доступны все системные функции после двух часов обучения работе с данной системой. После такого обучения среднее число ошибок оператора не должно превышать двух за рабочий день.

В идеале нефункциональные требования должны выражаться через количественные показатели, которые можно объективно измерить. В табл. 5.2 приведены показатели, с помощью которых можно специфицировать нефункциональные системные свойства.

На практике выразить нефункциональные требования с помощью количественных показателей весьма затруднительно. Часто заказчик ПО не может оформить свое видение будущей системы посредством требований, выраженных количественными показателями. Либо некоторые системные требования, например удобство сопровождения, вообще нельзя выразить через количественные показатели. Кроме того, затраты на объективное измерение количественных нефункциональных требований могут оказаться крайне высо-

ками. Поэтому часто документ, специфицирующий требования к системе, содержит описание системных целей совместно с четко сформулированными требованиями. Эти системные цели полезны, поскольку отражают представления (и приоритеты) заказчика о будущей системе. Вместе с тем заказчик должен понимать, что его системные цели могут трактоваться различными способами и их невозможно объективно проконтролировать.

Таблица 5.2. Количественные показатели для нефункциональных требований

Показатель	Единицы измерения
Скорость	Количество выполненных транзакций в секунду;
	время реакции на действия пользователя;
	время обновления экрана
Размер	Килобайты;
	количество модулей памяти
Простота эксплуатации	Время обучения персонала;
	количество статей в справочной системе
Надежность	Средняя продолжительность времени между двумя последовательными проявлениями ошибок в системе;
	вероятность выхода системы из строя;
	коэффициент готовности системы
Устойчивость к сбоям	Время восстановления системы после сбоя;
	процент событий, приводящих к сбоям;
	вероятность порчи данных при сбоях
Переносимость	Процент машинно-зависимых операторов;
	количество машинно-зависимых подсистем

Нефункциональные требования часто вступают в конфликт с другими требованиями, предъявляемыми системе. Например, в соответствии с одним из системных требований размер системы не должен превышать 4 Мбайт, поскольку она должна полностью поместиться в постоянное запоминающее устройство ограниченной емкости. Другое требование обязывает использовать для написания системы язык программирования Ada, который часто применяется для создания критических систем реального времени. Но, допустим, откомпилированная системная программа, написанная на языке Ada, занимает более 4 Мбайт. Итак, одновременное выполнение этих требований невозможно. В этой ситуации следует отказаться от одного из требований. Можно или применить другой язык программирования, или увеличить объем памяти, выделяемый для системы.

В принципе функциональные и нефункциональные требования в документе, описывающем требования к системе, должны быть разнесены по разным разделам. Но на практике это условие выполнить непросто. Если нефункциональные требования поместить отдельно от функциональных, будет трудно проследить взаимосвязи между ними. Если все требования собраны в одном списке, сложно провести анализ функциональных и нефункциональных требований в отдельности и определить требования, относящиеся к системе в целом. Вид представления требований в одном документе также существенно зависит от типа специфицируемой системы. Но в любом случае должны быть выделены требования, описывающие интеграционные свойства системы. Для этого их можно поместить в отдельный раздел либо каким-нибудь другим способом отделить от остальных требований.

5.1.3. Требования предметной области

Эти требования отображают условия, в которых будет эксплуатироваться программная система. Они могут быть представлены в виде новых функциональных требований, в виде ограничений на уже сформулированные функциональные требования или в виде указаний, как система должна выполнять вычисления. Эти требования очень важны, поскольку отображают ту предметную область, где будет использоваться данная система. Невыполнение требований предметной области может привести к выходу системы из строя.

В качестве примера рассмотрим требования к библиотечной системе (см. раздел 5.1.1).

1. Стандартный пользовательский иитерфейс, предоставляющий доступ ко всем библиотечным базам данных, должен основываться на стандарте Z39.50.
2. Для обеспечения авторских прав некоторые документы должны быть удалены из системы сразу после получения. Для этого, в зависимости от желания пользователя, эти документы могут быть распечатаны или на локальном системном сервере, или на сетевом принтере.

Первое требование является ограничением на системное функциональное требование. Оно указывает, что пользовательский интерфейс к базам данных должен быть реализован согласио соответствующему библиотечному стандарту. Второе требование является внешним и направлено на выполнение закона об авторских правах, применяемого к библиотечным материалам. Из этого требования вытекает, что система должна иметь средство "удалить_на_печать", примеияемое автоматически для некоторых типов библиотечных документов.

Во врезке 5.3 приведен пример требования предметной области, указывающего, как должны выполняться вычисления. Оно взято из спецификации системы автоматического торможеиия поезда. Эта система должна автоматически останавливать поезд на красный сигнал семафора. Данное требование указывает способ вычисления скорости поезда при торможении. Здесь использована терминология, применяемая при расчетах скоростей поезда. Чтобы разобраться в ней, необходимы соответствующие знания о системах управления поездами и их характеристиках.

Врезка 5.3. Пример требований предметной области

Торможение поезда вычисляется по формуле

$$D_{поезд} = D_{управление} + D_{градиент},$$

где $D_{градиент}$ равен 9.81 м·с^2 * компенсирующий градиент/альфа. Значение 9.81 м·с^2/альфа известно для всех типов поездов.

Приведенный пример показывает основную проблему, связанную с требованиями предметной области. Требования этого типа используют язык и обозначения, присущие данной предметной области, что затрудняет их понимание разработчиками ПО. Вследствие этого требования предметиой области не всегда выполняются так, как подразумевается заказчиками программной системы.

5.2. Пользовательские требования

Пользовательские требования к системе должны описывать функциональные и нефункциональные системные требования так, чтобы они были понятны даже пользователю, не имеющему специальиых технических знаний. Эти требования должны определять только

внешнее поведение системы, избегая по возможности определения структурных характеристик системы. Пользовательские требования должны быть написаны естественным языком с использованием простых таблиц, а также наглядных и понятных диаграмм.

Вместе с тем при описании требований на естественном языке могут возникнуть различные проблемы.

1. *Отсутствие четкости изложения.* Иногда нелегко изложить какую-либо мысль естественным языком четко и недвусмысленно, не сделав при этом текст многословным и трудночитаемым.
2. *Смешение требований.* В пользовательских требованиях отсутствует четкое разделение на функциональные и нефункциональные требования, на системные цели и проектную информацию.
3. *Объединение требований.* Несколько различных требований к системе могут описываться как единое пользовательское требование.

В качестве иллюстрации к описанным проблемам рассмотрим требование к среде программирования на языке Ada, представленное во врезке 5.4. Это требование содержит описание как общего плана, так и детализированное. Из информации, содержащейся в описании общего плана, следует, что средства управления конфигурацией являются составной частью интерфейса APSE, в то время как из более детализированного описания вытекает, что средства управления конфигурацией должны предоставлять доступ к объектам, входящим в состав групп, без указания их полных имен. Эту информацию лучше поместить в спецификацию системных требований.

Врезка 5.4. Требование к базе данных для среды программирования Ada

4.А.5. База данных должна поддерживать генерацию и управление конфигурацией объектов; в базе данных сгруппированные объекты могут выступать в виде отдельных объектов. Средство управления конфигурацией должно предоставить возможность доступа к объектам, входящим в состав групп, с помощью их неполных имен.

В документе, содержащем требования к системе, желательно отделять пользовательские требования от более детализированных системных требований. Иначе неподготовленный читатель пользовательских требований может "потонуть" в технических подробностях, понимание которых требует определенных профессиональных знаний. Пример смешения различных требований демонстрирует врезка 5.5. Он представляет требование к CASE-средству для редактирования схем структур программных систем. В этом требовании речь идет о возможности отображения или сокрытия сетки, помогающей пользователю точно позиционировать структурные элементы схемы.

Врезка 5.5. Пример пользовательского требования

2.6. Сфйихцжт тцтефдкйсмя хйцом. Для точного позиционирования структурных элементов схемы пользователь может отобразить на экране сетку, параметры которой могут задаваться (в сантиметрах или дюймах) посредством специальной опции на панели управления. По умолчанию сетка не отображается. Сетка может быть выведена на экран или скрыта в любой момент сессии редактирования, также в любой момент имеется возможность перехода с сантиметров на дюймы и обратно. Шаг сетки должен подгоняться под размер схемы.

В этом требовании переплетается не менее трех различных требований.

1. Концептуальное функциональное требование: система редактирования должна располагать возможностью отображения сетки. Это основная причина появления данного требования.
2. Нефункциональное требование, дающее подробную информацию о том, в каких единицах будет измеряться шаг сетки (сантиметры или дюймы).
3. Нефункциональное пользовательское требование, относящееся к интерфейсу: как пользователь может отобразить или скрыть сетку.

Описанное требование содержит и другую информацию, в частности необходимую для инициализации системы. В требовании сказано, что по умолчанию сетка отключена. Вместе с тем ничего не сказано о том, какие единицы измерения выбраны по умолчанию. Далее сказано, что пользователь может переключаться между сантиметрами и дюймами, но не сказано, что он может менять шаг сетки.

Когда пользовательское требование содержит так много информации, это затрудняет его понимание и ограничивает свободу разработчика в поиске решения задачи, поставленной в требовании. Пользовательские требования должны просто описывать основные возможности системы. Во врезке 5.6 показано переписанное мною пользовательское требование, где я сфокусировал внимание только на самом средстве отображения сетки без детализации его свойств.

Врезка 5.6. Описание средства отображения сетки

2.6. Средство отображения сетки

2.6.1. Редактор должен иметь средство вывода на экран сетки, которая состоит из параллельных горизонтальных и вертикальных линий и должна отображаться в виде фона на экране редактора. Сетка — пассивный элемент, облегчающий выравнивание пользователем структурных элементов схем.

Обоснование. Сетка должна помочь пользователю создать аккуратную схему с правильно размещенными элементами. Хотя активная сетка (такая, в которой элементы "привязываются" к узлам сетки) также может быть полезной, она не обеспечивает точного позиционирования элементов. Пользователь определит положение элементов схемы лучше, чем это сделает автоматизированное средство.

Спецификация: Эклипс/АРМ/Средства/DE/FS. Раздел 5.6

Обратите внимание на обоснование требования. Оно помогает разработчикам понять, почему это требование включено в спецификацию и в какой степени оно может измениться в будущем. Например, в обосновании требования на средство отображения сетки сказано, что также может быть полезной активная сетка, где элементы схемы автоматически "привязываются" к узлам сетки. Однако здесь предпочтение сознательно отдано пассивной сетке. Если в дальнейшем возникнет необходимость внести изменения в данное требование, то из этого обоснования будет видно, что вариант пассивной сетки выбран намеренно, а не появился на этапе реализации системы.

Следующий пример требования, также относящегося к системе редактирования схемы структуры ПО, показан во врезке 5.7. Это детализированная спецификация системной функции. В данном случае требование включает список действий, которые должен выполнить пользователь для реализации данной функции; иногда необходимо записать подобную последовательность действий, поскольку некоторые функции должны выполняться только строго определенным способом. Информация о том, как реализуется данная функция, в этом требовании отсутствует.

Врезка 5.7. Пользовательское требование по созданию структурных элементов схемы

3.5.1. Добавление структурных элементов в схему

3.5.1.1. Редактор должен иметь средство, предоставляющее пользователю возможность добавлять в схему новые структурные элементы выбранного типа.

3.5.1.2. Последовательность действий пользователя для добавления в схему нового структурного элемента:

1. **Пользователь выбирает тип добавляемого элемента.**
2. **Пользователь помещает курсор в нужную позицию на схеме и указывает, каким символом будет отображаться новый элемент.**
3. **Пользователь перемещает символ элемента в конечную позицию.**

Обоснование. **Такой подход к реализации функции добавления новых структурных элементов предоставляет пользователю непосредственный контроль над выбором типа элемента и его позиционированием на схеме.**

Спецификация: **Эклипс/АРМ/Средства/DE/FS. Раздел 3.5.1.**

Чтобы свести к минимуму неясности при написании пользовательских требований, я рекомендую придерживаться приведенных ниже правил.

1. Разработайте стандартную форму для записи пользовательских требований и неукоснительно ее придерживайтесь. Стандартная форма записи уменьшает неясности в формулировке требований и позволяет легко их проверить. Я рекомендую включать в форму записи требования не только саму его формулировку, но его обоснование и ссылку на более детализированную спецификацию требований.
2. Делайте различие между обязательными и описательными требованиями, как показано во врезке 5.7. Здесь обязательным требованием является наличие средства добавления новых структурных элементов, описательным — описание последовательности действий пользователя. Описательное требование не является абсолютно необходимым для реализации данного пользовательского требования и при необходимости может быть изменено.
3. Используйте разные начертания шрифта (полужирное и курсив) для выделения ключевых частей требования.
4. Избегайте по возможности компьютерного жаргона. Это не исключает использования технических терминов той предметной области, для которой разрабатывается программное обеспечение.

5.3. Системные требования

Системные требования — это более детализированное описание пользовательских требований. Они обычно служат основой для заключения контракта на разработку программной системы и поэтому должны представлять максимально полную спецификацию системы в целом. Системные требования также используются в качестве отправной точки на этапе проектирования системы.

Спецификация системных требований может строиться на основе различных системных моделей, таких, как объектная модель или модель потоков данных. Различные модели, используемые при разработке спецификации системных требований, описаны в главе 7.

В принципе системные требования определяют, что должна делать система, не показывая при этом механизма ее реализации. Но, с другой стороны, для полного описания системы требуется детализированная информация о ней, которая по возможности должна включать всю информацию о системной архитектуре. На то существует ряд причин.

1. Первоначальная архитектура системы помогает структурировать спецификацию требований. Системные требования должны описывать подсистемы, из которых состоит разрабатываемая система.
2. В большинстве случаев разрабатываемая система должна взаимодействовать с уже существующими системами. Это накладывает определенные ограничения на архитектуру новой системы.
3. В качестве внешнего системного требования может выступать условие использования для разрабатываемой системы специальной архитектуры (см. главу 18).

Спецификации системных требований часто пишутся естественным языком. Но, как указывалось в разделе 5.2, использование естественного языка может породить определенные проблемы при написании детализированной спецификации. Применение естественного языка подразумевает, что те, кто пишет спецификацию, и те, кто ее читает, одни и те же слова и выражения понимают одинаково. Однако на самом деле это не так, поскольку естественному языку присуща определенная размытость понятий. Вследствие этого одно и то же требование может трактоваться разными людьми по-разному.

Чтобы избежать подобных проблем, разработаны методы описания требований, которые структурируют спецификацию и уменьшают размытость определений. Эти методы представлены в табл. 5.3. Кроме этого, разработаны другие подходы, например специальные языки описания требований [333, 9, 35, 10, 18*, 19*], которые используются относительно редко. В этой главе рассматриваются первые два подхода из описанных в табл. 5.3.

Таблица 5.3. Способы записи спецификаций требований

Система записи	Описание
Структурированный естественный язык	Использование стандартных форм и шаблонов для написания спецификации
Языки описания программ	Использование специальных структурированных языков, подобных языкам программирования, где спецификация требований строится на основе выбранной операционной модели системы
Графические нотации	Графический язык, использующий для описания функциональных требований диаграммы и блок-схемы, дополненные текстовыми пояснениями. Наиболее известный пример такого графического языка – диаграммы структурного анализа и проектирования ПО (SADT) [299, 308, 7*]. В следующей главе рассматривается другой пример графических нотаций, а именно метод описания вариантов использования
Математические спецификации	Это системы нотаций, основанные на математических концепциях, таких, как теория конечных автоматов или теория множеств. Это формализованная однозначная и лишенная двусмысленности запись системных требований. Однако многие заказчики ПО не понимают формальных спецификаций, вследствие чего возникают определенные проблемы при заключении контрактов на разработку программных продуктов. Формальные спецификации рассматриваются в главе 9

5.3.1. Структурированный язык спецификаций

Это сокращенная форма естественного языка, предназначенная для написания спецификации требований. Достоинством такого подхода к написанию спецификаций является то, что он сохраняет выразительность и понятность естественного языка и вместе с тем формализует описание требований. Структурированность языка проявляется в использовании специальной терминологии, а также шаблонов для описания системных требований. Структурированный язык может включать языковые конструкции, взятые из языков программирования.

Пример использования структурированного языка для описания системных требований приведен в работе [160]. Здесь для написания спецификации требований для программной системы управления полетами разработаны специальные формы, помогающие описать входные и выходные данные и функции системы.

Для описания системных требований часто разрабатываются специальные формы и шаблоны. Они должны учитывать, на основе чего строится спецификация: на основе объектов, управляемых системой, на основе функций, выполняемых системой, или на основе событий, обрабатываемых системой. Пример формы для спецификации показан во врезке 5.8. Это более детализированное описание функции создания структурных элементов для системы редактирования программных архитектур, описанной во врезке 5.7.

Врезка 5.8. Спецификация системного требования, использующая стандартную форму

Эклипс/АРМ/Средства/DE/RD/3.5.1

Функция. Добавление структурных элементов в схему.

Описание. Добавление структурных элементов в существующую схему системной архитектуры. Пользователь выбирает тип структурного элемента и его местоположение. После вставки в схему структурный элемент становится выделенным (текущим структурным элементом). Пользователь определяет местоположение элемента путем перемещения курсора по области схемы.

Входные данные. Тип элемента, позиция элемента, идентификатор схемы.

Источники входных данных. Тип элемента и позиция элемента задаются пользователем, идентификатор схемы получен из базы данных проекта.

Выходные данные. Идентификатор схемы.

Пункт назначения. База данных проекта. Идентификатор схемы помещается в базу данных проекта по завершении выполнения данной функции.

Для выполнения функции требуется схема, определенная входным идентификатором схемы.

Предусловие. Схема открыта и отображается на экране пользователя.

Постусловие. Схема, за исключением вставки нового структурного элемента, не изменяется.

Побочные эффекты. Нет.

Спецификация: Эклипс/АРМ/Средства/DE/RD/3.5.1

Стандартные формы, используемые для специфицирования функциональных требований, должны содержать следующую информацию.

1. Описание функции или объекта.
2. Описание входных данных и их источники.

3. Описание выходных данных с указанием пункта их назначения.
4. Указание, что необходимо для выполнения функции.
5. Если это спецификация функции, необходимо описание предварительных условий (предусловий), которые должны выполняться перед вызовом функции, и описание заключительного условия (постусловия), которое должно быть выполнено после завершения выполнения функции.
6. Описание побочных эффектов (если они есть).

Использование структурированного языка снимает некоторые проблемы, присущие спецификациям, написанным естественным языком, поскольку снижает "вариабельность" спецификации и более эффективно ее структурирует. Вместе с тем некоторая "размытость" определений и описаний в спецификации остается. Альтернативой использованию структурированного естественного языка может служить специальный язык описания спецификаций (рассмотрен в следующем разделе), который полностью снимает проблему нечеткости описания требований. Но с другой стороны, неспециалист найдет такую спецификацию трудной для чтения и понимания.

5.3.2. Создание спецификаций с помощью PDL

Для уменьшения присущей естественному языку нечеткости понятий при описании системных требований используется специальный язык описания программ (program description language – PDL). Этот язык подобен таким языкам программирования, как Java и Ada, но более абстрактен. Достоинством применения PDL для создания спецификации является то, что в такой спецификации можно проверить синтаксис и семантику существующими программными средствами. Эти проверки позволяют удалить ошибки и несогласованность в описании требований.

Применение PDL приводит к очень подробным и детализированным спецификациям, которые иногда просто невозможно ввести в заключительный документ с описанием системных требований. Я рекомендую использовать PDL в следующих ситуациях.

1. Если описываемая операция состоит из последовательности простых действий и важен порядок их выполнения. Описать такие последовательности действий на естественном языке порой затруднительно, поскольку их выполнение может сопровождаться вложенными условиями или они могут повторяться циклически. Этой ситуации соответствует спецификация системы обслуживания банкоматов, приведенная в листинге 5.1. Здесь в качестве PDL я использовал язык Java. В этом листинге я привел только часть спецификации, опустив описания некоторых сервисов. Полную спецификацию можно найти на Web-странице данной книги.
2. Если необходимо специфицировать аппаратные или программные интерфейсы, так как практически во всех спецификациях системных требований приходится описывать интерфейсы.

Листинг 5.1. Спецификация системы управления банкоматами

```
class ATM {
    // декларативная часть
    public static void main (String args[]) throws InvalidCard{
        try {
            thisCard.read () ;
                //может породить исключительную ситуацию InvalidCard
            pin = KeyPad.readPin () ; attempts = 1 ;
```

```
            while ( !thisCard.pin.eguals (pin) & attempts < 4 )
                { pin = KeyPad.readPin () ; attempts = attempts + 1 ;
                }
                if ( !thisCard.pin.eguals (pin) )
                    throws new InvalidCard ("Неправильный  Pin-код") ;
            thisBalance = thisCard.getBalance ();
            do ( Screen.promt ("Пожалуйста, выберите сервис");
                service = Screen.touchKey ();
                switch (service) {
                    case Services.withdrawalWithReceipt:
                        receiptRequired = true ;
                    case Services.withdrawalNoReceipt:
                        amount = KeyPad.readAmount () ;
                        if (amount > thisBalance)
                        { Screen.printmsg ("Счет превышен") ;
                            break ;
                        }
                        Dispenser.deliver (amount) ;
                        newBalance = thisBalance - amount ;
                        if (receiptRequired)
                            Receipt.print (amount, newBalance) ;
                        break ;
                    //далее описание других сервисов
                    default: break ;
                }
            }
            while (service != Services.quit) ;
            thisCard.returnToUser ("Пожалуйста, заберите карточку");
        }
        catch (InvalidCard e )
        { Screen.printmsg ("Недействительна карточка или Pin-код") ;
        }
        //далее обработка других исключений
    }//main ()
}//ATM
```

Если читатель системной спецификации знаком с PDL, чтение такой спецификации не вызовет у него затруднений. Кроме того, если PDL построен на основе какого-либо языка программирования, легко преобразовать спецификацию в системную архитектуру, при этом возможность неверной интерпретации требований сведена к минимуму.

Вместе с тем подход к построению спецификаций, основанный на PDL, имеет свои недостатки.

1. PDL, используемый для написания спецификации, может не иметь достаточных средств для описания всех системных функций.
2. Спецификации, созданные с помощью PDL, понятны только людям, имеющим определенные знания языков программирования.
3. Системная архитектура, полученная на основе такой спецификации, не является системной моделью, которая помогла бы пользователю разобраться в структуре системы.

Наиболее эффективным способом разработки спецификаций является сочетание подхода, основанного на PDL, с использованием структурированного естественного языка. В этом случае формализованные записи на естественном языке используются для описания системы в целом, а PDL – для описания последовательностей управляющих действий или для детализированного описания интерфейсов.

5.3.3. Спецификация интерфейсов

Подавляющее большинство разрабатываемых программных систем должны взаимодействовать с другими, уже существующими системами. Для того чтобы новая система могла работать совместно с другими системами, необходимо специфицировать интерфейсы этих систем. Такие спецификации создаются на раннем этапе разработки систем и включаются (обычно в виде приложения) в спецификацию системных требований.

Различают три типа специфицируемых интерфейсов.

1. Процедурные интерфейсы, когда существующие подсистемы предлагают набор сервисов, доступных посредством вызываемой интерфейсной процедуры.
2. Структуры (интерфейсные форматы) данных, которые пересылаются от одной подсистемы к другой. В этом случае может использоваться PDL, основанный на языке программирования Java, который позволяет описать классы данных с соответствующими полями, формирующими структуру данных. Однако, я полагаю, для описания этого типа интерфейса наиболее подходят диаграммы "сущность-связь", описанные в главе 7.
3. Специальные представления данных, например в виде упорядоченной последовательности двоичных разрядов. Язык Java не поддерживает такие детальные описания данных, поэтому я не рекомендую в подобном случае использовать PDL, основанный на языке Java.

Формальные нотации, приведенные в главе 9, позволяют описать интерфейсы четко и недвусмысленно. Однако такие спецификации понятны только специалистам, обладающим определенными знаниями. Формальные нотации редко используются на практике, хотя, с моей точки зрения, они идеально подходят для создания спецификаций интерфейсов. Менее формальные спецификации интерфейсов, полученные с помощью PDL, являются компромиссом между понятностью и точностью описания интерфейсов.

В листинге 5.2 представлен пример описания интерфейса первого типа (из перечисленных выше). Это описание процедурного интерфейса сервера печати. Он управляет очередями на печать, осуществляемую несколькими принтерами. Пользователи могут проверять очередь, соответствующую любому принтеру, и удалять свои файлы из очереди. Они также могут переключать свои файлы с одного принтера на другой.

Листинг 5.2. Описание интерфейса сервера печати с помощью PDL

```
interface PrintServer {
    // определение абстрактного сервера печати
    // требуется: интерфейс Printer, интерфейс PrintDoc
    // предоставляет функции: initialize (инициализация),
    // print (печать),
    // displayPrintQueue (отображение очереди на печать),
    // cancelPrintJob (удаление файла из очереди),
    // switchPrinter (переключение между принтерами)

        void initialize ( Printer p ) ;
        void print ( Printer p, PrintDoc d ) ;
        void displayPrintQueue ( Printer p ) ;
        void cancelPrintJob ( Printer p, PrintDoc d ) ;
        void switchPrinter ( Printer p1, Printer p2, PrintDoc d ) ;
    }//PrintServer
```

Спецификация, приведенная в листинге 5.2, – абстрактная модель сервера печати без детализации интерфейсных частностей. Интерфейсные функции можно описать с помощью структурированного естественного языка (как показано во врезке 5.6), посредством PDL, основанного на языке программирования Java (см. листинг 5.1), либо с помощью формальных нотаций, которые описаны в главе 9.

5.4. Документирование системных требований

Документ, содержащий требования, также называемый спецификацией системных требований, – это официальное предписание для разработчиков программной системы. Он содержит пользовательские требования и детализированное описание системных требований. В некоторых случаях пользовательские и системные требования могут не различаться, выступая совместно в виде однородного описания системы. В других случаях пользовательские требования приводятся во введении документа-спецификации. Если общее количество требований велико, детализированные системные требования могут быть представлены в виде отдельного документа.

Системную спецификацию читает множество разных людей, начиная от высшего руководства компании – заказчика системы и заканчивая рядовым разработчиком системы. На рис. 5.3, взятом из статьи [204], показаны категории читателей спецификации.

Рис. 5.3. Читатели системной спецификации

Хенингер (Heninger, [160]) сформулировал шесть условий, которым должна соответствовать спецификация программной системы.

- Описывать только внешнее поведение системы.
- Указывать ограничения, накладываемые на процесс реализации системы.
- Предусматривать возможность внесения изменений в спецификацию.
- Служить справочным средством в процессе сопровождения системы.
- Отображать весь жизненный цикл системы.
- Предусматривать реакцию системы и группы сопровождения на непредвиденные (нештатные) ситуации.

Хотя со времени написания этих условий прошло более 20 лет, они не утратили своей актуальности и являются хорошим подспорьем при разработке спецификаций. Вместе с тем подчас сложно или даже невозможно описать систему только в терминах внешнего поведения, т.е. только посредством функций, которые она должна выполнять, поскольку в спецификации также должны быть отражены ограничения, накладываемые окружением уже существующих систем. Другое условие — охват всего жизненного цикла системы — принимается всеми разработчиками, но редко выполняется на практике.

Многие организации, такие, как Министерство обороны США и Институт инженеров по электротехнике и радиоэлектронике IEEE, разработали собственные стандарты документирования спецификаций. В монографии [89] описаны некоторые из этих стандартов, а также проведено их сравнение. Наиболее известный стандарт разработан IEEE и называется IEEE/ANSI 830-1993 [IEEE, 1993]. В книге [334], содержащей великолепный обзор работ по технологии разработки требований, приведено полное описание этого стандарта. Данный стандарт предполагает следующую структуру спецификации.

1. **Введение**
   1.1. Цели документа
   1.2. Назначение программного продукта
   1.3. Определения, акронимы и аббревиатуры
   1.4. Список литературы и других источников
   1.5. Обзор спецификации
2. **Общее описание**
   2.1. Описание программного продукта
   2.2. Функции программного продукта
   2.3. Пользовательские характеристики
   2.4. Общие ограничения
   2.5. Обоснования, предположения и допущения
3. **Спецификация требований** охватывает функциональные, нефункциональные и интерфейсные требования. Это наиболее значимая часть документа, но вследствие крайне широкого диапазона возможных требований, предъявляемых программным системам, в стандарте не определена структура этого раздела. Здесь могут быть документированы внешние интерфейсы, описаны функциональные возможности системы, приведены требования, определяющие логическую структуру баз данных, ограничения, накладываемые на структуру системы, описаны интеграционные свойства системы или ее качественные характеристики.

4. **Приложения**
5. **Указатели**

Хотя стандарт IEEE не идеален, он может служить отправной точкой при написании спецификации. Конечно, при ее написании необходимо также учитывать стандарты, принятые в организации – разработчике ПО. В табл. 5.4 описаны возможные разделы спецификации, построенной на основании стандарта IEEE. Я также включил раздел о возможной эволюции системы, как рекомендует Хенингер.

Таблица 5.4. Структура спецификации требований

Раздел	Описание
Предисловие	Здесь определяется круг лиц, на которых рассчитан данный документ. Описываются предыдущие версии разрабатываемого программного продукта, а также изменения, внесенные в каждую версию. Дается обоснование для создания новой версии продукта
Введение	Здесь более развернуто обосновывается необходимость создания системы. Кратко перечисляются системные функции и объясняется, как система будет работать совместно с другими системами. Должно быть показано, как разработка системы "вписывается" в общую бизнес-стратегию компании, заказывающей программный продукт
Глоссарий	Дается описание технических терминов, используемых в документе. Здесь не делается каких-либо предположений об уровне знаний или практическом опыте читателя документа
Пользовательские требования	Описываются сервисы, предоставляемые пользователям, и нефункциональные системные требования. Это описание может быть сделано на естественном языке с использованием диаграмм, блок-схем и других форм записи, понятных заказчику программной системы. Здесь также должны быть приведены стандарты на программный продукт и процесс его разработки
Системная архитектура	Здесь приводится высокоуровневое представление возможной системной архитектуры с указанием, как распределены системные функции по компонентам системы. Обязательно должны быть выделены повторно используемые (т.е. уже существующие) компоненты
Системные требования	Подробно описываются функциональные и нефункциональные требования. Если необходимо, нефункциональные требования дополняются описанием интерфейсов других систем
Системные модели	Здесь представлено несколько системных моделей, показывающих взаимоотношения между системными компонентами и между системой и ее окружением. Это могут быть объектные модели, модели потоков данных или модели данных
Эволюция системы	Приводятся основные предположения и допущения, на которых базируется система, а также ожидаемые (прогнозируемые) изменения в аппаратных средствах, в потребностях пользователей и т.п.

Окончание табл. 5.4

Раздел	Описание
Приложения	Здесь приводится специализированная информация, относящаяся к разрабатываемой системе, например описание аппаратных средств или базы данных, с которыми должна работать система. При описании аппаратных средств необходимо показать минимальную и оптимальную конфигурации, при которых может работать программная система. Описание базы данных должно отображать логическую структуру данных, с которыми будет работать система, и отношения между ними
Указатели	В документе возможно использование различных указателей. Это может быть обычный алфавитный указатель, указатель диаграмм или указатель системных функций

Конечно, информация, включаемая в спецификацию, зависит от типа разрабатываемого программного обеспечения и от выбранной технологии разработки. Например, если при разработке будет использоваться эволюционный подход, многие разделы спецификации, перечисленные в табл. 5.4, будут излишни. Если же разрабатываемое ПО является только частью большой системы, состоящей из взаимодействующих аппаратных и программных систем, тогда по необходимости требования будут очень подробными и детализированными. В этом случае спецификация может иметь значительный объем и будет содержать больше разделов, чем перечислено в табл. 5.4. Для больших документов необходимо делать оглавление и подробные указатели для поиска необходимой информации.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Требования программной системы — это описание того, что система должна делать, а также ограничений, накладываемых на ее поведение и реализацию.
- Функциональные требования — описание сервисов, предоставляемых системой, и способов выполнения вычислительных операций. Требования предметной области — это функциональные требования, которые вытекают из характеристик той предметной области, где будет эксплуатироваться разрабатываемая система.
- Нефункциональные требования — это ограничения, накладываемые на систему, на процесс разработки системы, а также внешние требования. Они описывают свойства системы в целом.
- Пользовательские требования предназначены для людей, которые будут эксплуатировать систему. Они должны писаться естественным языком с использованием таблиц и диаграмм, простых для восприятия.
- Системные требования должны максимально точно описывать функции, выполняемые системой. Для уменьшения неточности формулировок системные требования могут записываться с помощью структурированных языков. Это может быть структурированная форма естественного языка, язык, построенный на основе какого-нибудь языка программирования высокого уровня, либо специальный язык для специфицирования требований.
- Спецификация требований — это официальное изложение системных требований. Этот документ строится таким образом, чтобы его могли использовать как заказчики программного продукта, так и разработчики ПО.

Упражнения

5.1. Обсудите проблемы, возникающие при формулировании пользовательских и системных требований на естественном языке, и покажите на небольших примерах, как структурированный естественный язык помогает избежать этих проблем.

5.2. Исследуйте точность формулировок следующего требования, описывающего автоматизированную систему продажи билетов.

5.3. Автоматизированная система продажи железнодорожных билетов. Пользователь указывает пункт назначения, вставляет кредитную карточку и вводит личный идентификационный номер. Система выдает проездной билет и снимает с кредитной карточки сумму, равную стоимости проезда в указанный пункт. Когда пользователь нажимает начальную кнопку, отображается меню возможных пунктов назначения с указанием стоимости проезда до каждого пункта. После выбора пункта назначения пользователю предлагается вставить кредитную карточку. Затем проверяется кредитная карточка, после чего пользователю предлагается ввести личный идентификационный номер. По окончании операций с кредитной карточкой выдается проездной билет.

5.4. Перепишите требование предыдущего пункта, используя структурный подход, описанный в этой главе.

5.5. Запишите системные требования для системы продажи билетов с помощью языка, основанного на языке программирования Java. Обратите внимание на обработку ошибок пользователя.

5.6. Опишите три типа нефункциональных требований, которые могут иметь место в программных системах. Приведите примеры каждого типа требований.

5.7. Запишите нефункциональные требования для описанной выше автоматизированной системы продажи билетов, характеризующие надежность системы и время ее ответа.

5.8. Какими свойствами должен обладать язык программирования, чтобы его можно было применить для написания спецификаций интерфейсов? В этом аспекте рассмотрите возможности языков C, Java и Ada.

5.9. Поясните, как в спецификации системных требований можно проследить взаимосвязь между функциональными и нефункциональными требованиями.

5.10. Вы устроились на работу в фирму, которая заказала разработать некую программную систему в той организации, где вы работали ранее. Вы обнаружили, что фирма-заказчик интерпретирует некоторые системные требования не так, как организация-разработчик. Подумайте, что вы должны делать в такой ситуации. Вы знаете, что стоимость программной системы возрастет, если не будет устранено разночтение системных требований. С другой стороны, вы связаны обязательством неразглашения сведений о своей предыдущей работе.

6 Разработка требований

Цели

Цель настоящей главы — описать процесс разработки требований к проектируемой системе. Прочитав эту главу, вы должны:

- знать основные процессы разработки требований и отношения между ними;
- ознакомиться с методами формирования и анализа требований;
- понимать важность аттестации требований;
- знать, почему необходимо управление требованиями.

Содержание

Разработка требований – это процесс, включающий мероприятия, необходимые для создания и утверждения документа, содержащего спецификацию системных требований. Различают четыре основных этапа процесса разработки требований: анализ технической осуществимости создания системы, формирование и анализ требований, специфицирование требований и создание соответствующей документации, а также аттестация этих требований. В этой главе рассмотрены все перечисленные этапы, за исключением специфицирования и документирования, которые описаны в главе 5. На рис. 6.1 показаны взаимосвязи между этими этапами и документы, сопровождающие каждый этап процесса разработки системных требований.

Рис. 6.1. Процесс разработки требований

На рис. 6.1 показаны этапы формирования, документирования и проверки требований. Но поскольку в процессе разработки системы в силу разнообразных причин требования могут меняться, управление требованиями, т.е. процесс управления изменениями системных требований, является необходимой составной частью деятельности по их разработке. Управление требованиями описано в заключительном разделе главы.

Некоторые специалисты полагают, что для разработки требований необходимо применение структурных методов, например объектно-ориентированного анализа [302, 54, 12*, 25*]. Такой процесс разработки требований включает анализ системы и разработку набора графических моделей системы, которые затем используются для ее подробного специфицирования. Хотя структурные методы играют существенную роль в процессе разработки требований, многие аспекты этого процесса не могут быть охвачены данными методами. Они неэффективны на ранних стадиях процесса разработки, таких, как формирование требований. Поэтому здесь рассматриваются общие подходы к разработке требований, а структурные методы анализа и системные модели описываются в главе 7.

6.1. Анализ осуществимости

Для новых программных систем процесс разработки требований должен начинаться с анализа осуществимости. Началом такого анализа является общее описание системы и ее назначения, а результатом анализа – отчет, в котором должна быть четкая рекомендация, продолжать или нет процесс разработки требований проектируемой системы. Другими словами, анализ осуществимости должен осветить следующие вопросы.

1. Отвечает ли система общим и бизнес-целям организации-заказчика и организации-разработчика?
2. Можно ли реализовать систему, используя существующие на данный момент технологии и не выходя за пределы заданной стоимости?
3. Можно ли объединить систему с другими системами, которые уже эксплуатируются?

Критическим является вопрос, будет ли система соответствовать целям бизнеса. Если система не соответствует этим целям, она не представляет никакой ценности для бизнеса. В то же время многие организации разрабатывают системы, не соответствующие их целям, либо не совсем ясно понимая эти цели, либо под влиянием политических или общественных факторов.

Выполнение анализа осуществимости включает сбор и анализ информации о будущей системе и написание соответствующего отчета. Сначала следует определить, какая именно информация необходима, чтобы ответить на поставленные выше вопросы. Например, эту информацию можно получить, ответив на следующие вопросы.

1. Что произойдет с организацией, если система не будет введена в эксплуатацию?
2. Какие текущие проблемы существуют в организации и как новая система поможет их решить?
3. Каким образом система будет способствовать целям бизнеса?
4. Требует ли разработка системы технологии, которая до этого не использовалась в организации?

Как только будут сформулированы подобные вопросы, необходимо определить источники информации. Это могут быть менеджеры отделов, где система будет использоваться, разработчики программного обеспечения, знакомые с типом будущей системы, технологи, конечные пользователи и т.д.

После обработки собранной информации готовится отчет по анализу осуществимости создания системы. В нем должны быть даны рекомендации относительно продолжения разработки системы. Могут быть предложены изменения бюджета и графика работ по созданию системы или предъявлены более высокие требования к системе.

6.2. Формирование и анализ требований

После выполнения анализа осуществимости следующим этапом процесса разработки требований является формирование (определение) и анализ требований. На этом этапе команда разработчиков ПО работает с заказчиком и конечными пользователями системы для выяснения области применения, описания системных сервисов, определения режимов работы системы и ее характеристик выполнения, аппаратных ограничений и т.д.

В процесс формирования требований могут быть вовлечены люди разных профессий. В нем принимают участие конечные пользователи, которые будут работать с системой, инженеры, которые разрабатывают и эксплуатируют подобные системы, бизнес-менеджеры, специалисты по предметной области, где будет эксплуатироваться система, и даже представители профсоюзов.

Процесс формирования и анализа требований достаточно сложен по ряду причин.

1. Лица, участвующие в формировании требований, часто не знают конкретно, чего они хотят от компьютерной системы, за исключением наиболее общих положений; им трудно сформулировать, что они ожидают от системы; они могут предъявлять нереальные требования, так как не подозревают, какова стоимость их реализации.

2. Лица, участвующие в формировании требований, выражают в этих требованиях собственные точки зрения, основываясь на личном опыте работы.
3. Лица, участвующие в формировании требований, имеют различные предпочтения и могут выражать их разными способами. Разработчики должны определить все потенциальные источники требований и выделить общие и противоречивые требования.
4. На требования к системе могут влиять политические факторы. Они могут исходить от руководителей, которые предъявляют требования только для того, чтобы усилить свое влияние в организации.
5. Экономическая и бизнес-обстановка, в которой происходит формирование требований, неизбежно будет меняться в ходе выполнения этого процесса. Следовательно, и важность отдельных требований может изменяться. Новые требования могут быть выдвинуты новым лицом, с которым первоначально не консультировались.

Обобщенная модель процесса формирования и анализа требований показана на рис. 6.2. Каждая организация использует собственный вариант этой модели, зависящий от "местных" факторов: опыта работы коллектива разработчиков, типа разрабатываемой системы, используемых стандартов и т.д.

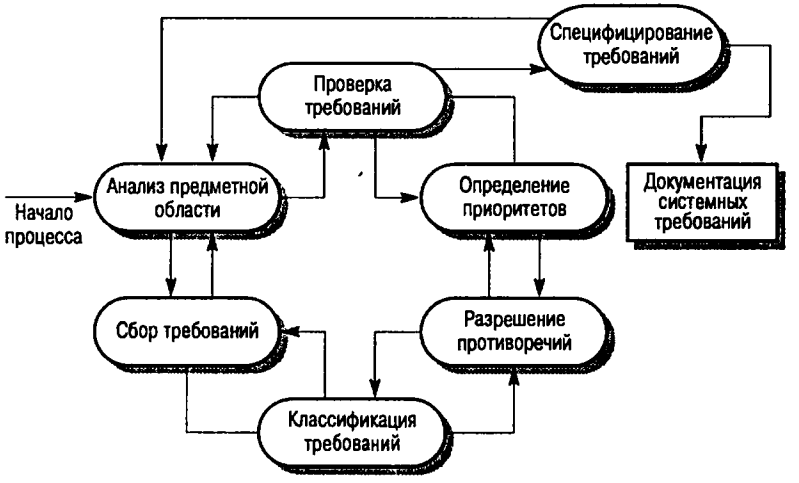


Рис. 6.2. Процесс формирования и анализа требований

Процесс формирования и анализа требований проходит через ряд этапов.

1. *Анализ предметной области.* Аналитики должны изучить предметную область, где будет эксплуатироваться система.
2. *Сбор требований.* Это процесс взаимодействия с лицами, формирующими требования. Во время этого процесса продолжается анализ предметной области.
3. *Классификация требований.* На этом этапе бесформенный набор требований преобразуется в логически связанные группы требований.

4. *Разрешение противоречий.* Без сомнения, требования многочисленных лиц, занятых в процессе формирования требований, будут противоречивыми. На этом этапе определяются и разрешаются противоречия такого рода.
5. *Назначение приоритетов.* В любом наборе требований одни из них будут более важны, чем другие. На этом этапе совместно с лицами, формирующими требования, определяются наиболее важные требования.
6. *Проверка требований.* На этом этапе определяется их полнота, последовательность и непротиворечивость.

Как показано на рис. 6.2, процесс формирования и анализа требований циклический, с обратной связью от одного этапа к другому. Цикл начинается с анализа предметной области и заканчивается проверкой требований. Понимание требований предметной области увеличивается в каждом цикле процесса формирования требований.

В этом разделе описаны три подхода к формированию требований: метод, основанный на множестве опорных точек зрения, сценарии и этнографический метод. Другие подходы, которые могут использоваться в процессе разработки требований, — это методы структурного анализа, рассматриваемые в главе 7, и методы прототипирования, описываемые в главе 8. Не существует универсального подхода к формированию и анализу требований. Обычно для разработки требований одновременно используется несколько подходов.

6.2.1. Опорные точки зрения

Любая средняя или большая система ПО обычно имеет различные типы конечных пользователей. Многие лица, участвующие в формировании требований, в своих требованиях к системе выражают собственные интересы. Например, в процессе формировании требований для системы банкоматов участвуют следующие лица.

1. *Обычные клиенты банка*, пользующихся услугами банкоматов.
2. *Представители других банков*, имеющих взаимные соглашения с данным банком о совместном использовании банкоматов.
3. *Менеджеры филиалов банка*, получающих информацию из системы управления банкоматами.
4. *Сотрудники филиалов банка*, вовлеченные в повседневную работу системы банкоматов, обрабатывающие рекламации клиентов и т.д.
5. *Администраторы баз данных*, ответственные за связь банкоматов с базой данных клиентов.
6. *Руководители службы безопасности банка*, обеспечивающей защиту системы банкоматов.
7. *Отдел маркетинга банка*, использующий систему банкоматов как средство маркетинга.
8. *Разработчики аппаратных и программных средств*, ответственные за сопровождение и модернизацию аппаратных и программных средств.

Этот список показывает, что даже для относительно простой системы существует много различных точек зрения, которые должны быть рассмотрены. Различные точки зрения на проблему позволяют увидеть ее с разных сторон. Однако эти взгляды не являются полностью независимыми и обычно перекрывают друг друга, а потому могут служить основой общих требований.

Подход с использованием различных *опорных* точек зрения к разработке требований признает эти различные (опорные) точки зрения и использует их в качестве ос-

новы построения и организации как процесса формирования требований, так и непосредственно самих требований. Сильная сторона анализа, ориентированного на различные опорные точки зрения, в том, что он признает множество взглядов и обеспечивает основу для обнаружения противоречий в требованиях, предложенных различными лицами.

Различные методы предлагают разные трактовки выражения "точка зрения". Точки зрения можно трактовать следующим образом[1].

1. *Как источник информации о системных данных.* В этом случае на основе опорных точек зрения строится модель создания и использования данных в системе. В процессе формирования требований отбираются все такие точки зрения, на их основе определяются данные, которые будут созданы или использованы при работе системы, и способы обработки этих данных. Методы SADT [299, 308, 7*] и CORE [242] используют эту интерпретацию точек зрения.
2. *Как структура представлений.* В этом случае точки зрения рассматриваются как особая часть модели системы [115, 260]. Например, на основе различных точек зрения могут разрабатываться модели "сущность–связь", модели конечного автомата и т.д.
3. *Как получатели системных сервисов.* В этом случае точки зрения являются внешними (относительно системы) получателями системных сервисов [202, 203]. Точки зрения помогают определить данные, необходимые для выполнения системных сервисов или их управления.

Каждая из этих интерпретаций точек зрения имеет сильные и слабые стороны. Опорные точки зрения как источники информации о системных данных и как представления имеют ценность для обнаружения противоречий в требованиях [101]. Но использовать их в процессе структурного анализа требований затруднительно, поскольку здесь не фиксируются связи между точками зрения и типами участников формирования требований.

Интерактивные системы поставляют сервисы конечным пользователям. Поэтому наиболее эффективным подходом к анализу таких систем является использование внешних опорных точек зрения. Эти точки зрения взаимодействуют с системой, получая от нее сервисы и продуцируя данные и управляющие сигналы.

Этот тип точек зрения имеет ряд преимуществ.

1. Точки зрения, внешние к системе, — естественный способ структурирования процесса формирования требований.
2. Сравнительно просто решить, какие точки зрения следует оставить в качестве опорных: они должны отображать какой-либо способ взаимодействия с системой.
3. Данный подход полезен для создания нефункциональных требований, с которыми можно связать какой-либо сервис. Многократно повторяющиеся разные точки зрения позволяют сформировать для сервисов различные нефункциональные требования.

---

[1] *Для понимания дальнейшего материала следует принять во внимание, что автор часто отождествляет точку зрения и физическое лицо – носителя этой точки зрения. Поэтому он "одушевляет" точки зрения и говорит, например, о взаимодействии точек зрения с системой или о точках зрения – получателях системных сервисов (см. далее). Другими словами, здесь термин "точка зрения" означает гораздо больше, чем просто мнение отдельного человека о чем-либо.* – Прим. ред.

На основе этого подхода разработан метод VORD (Viewpoint-Oriented Requirements Definition – определение требований на основе точек зрения) для формирования и анализа требований [203, 204]. Основные этапы метода VORD показаны на рис. 6.3.

1. Идентификация точек зрения, получающих системные сервисы, и идентификация сервисов, соответствующих каждой точке зрения.
2. Структурирование точек зрения – создание иерархии сгруппированных точек зрения. Общесистемные сервисы предоставляются более высоким уровням иерархии и наследуются точками зрения низшего уровня.
3. Документирование опорных точек зрения, которое заключается в точном описании идентифицированных точек зрения и сервисов.
4. Отображение системы точек зрения, которая показывает системные объекты, определенные на основе информации, заключенной в опорных точках зрения.

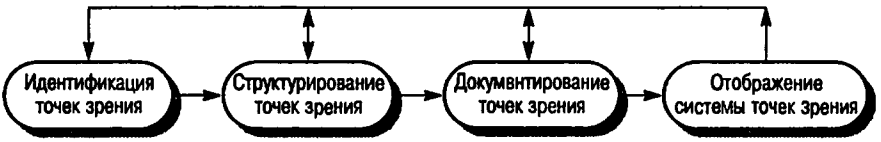


Рис. 6.3. Метод VORD

Точки зрения и информация о сервисах в методе VORD собираются с помощью стандартных форм (шаблонов точек зрения и шаблонов описания сервисов). В методе VORD также используются различные графические представления, включая диаграммы иерархии точек зрения, подобные показанным на рис. 6.7, и сценарии событий (пример сценария показан на рис. 6.8).

Я покажу использование метода VORD на первых трех шагах анализа требований для системы управления банкоматами. Банкомат имеет встроенное программное обеспечение для управления аппаратными средствами и для связи с центральной базой данных банковских счетов.

Банкомат принимает запросы клиента, выдает наличные деньги, информацию о счете, изменяет данные в базе данных банка и т.д. Банкоматы одиого банка могут позволить клиентам других банков использовать какую-то часть своих сервисов (обычно это снятие со счета наличных денег и запрос о текущем балансе счета).

Первым шагом в формировааии требований является идентификация опорных точек зрення. Во всех методах формирования требований, основанных на использовании точек зрения, начальная идеитификация является наиболее трудной задачей. Один из подходов к идентификации точек зрения – метод "мозговой атаки", когда определяются потенциальные системные сервисы и организации, взаимодействующие с системой. Организуется встреча лиц, участвующих в формировании требований, которые предлагают свои точки зрения. Эти точки зрения представляются в виде диаграммы, состоящей из ряда круговых областей, отображающих возможные точки зрения (рис. 6.4).

Во время "мозговой атаки" необходимо идентифицировать потенциальные опорные точки зрения, системные сервисы, входные данные, нефункциональные требования, управляющие события и исключительные ситуации. Источниками информации, которые можно использовать в создании этого начального образа системы, могут служить документы, описывающие назначение системы, знания инженеров-программистов из предыдущих проектов или опыт клиентов банка. Может быть проведен опрос менеджеров банка, обслуживающего персонала, консультантов, инженеров и клиентов.

Следующей стадией процесса формирования требований будет идентификация опорных точек зрения (на рис. 6.4 показаны в виде темных круговых областей) и сервисов (показаны в виде затененных областей). Сервисы должны соответствовать опорным точкам зрения. Но могут быть сервисы, которые не поставлены им в соответствие. Это означает, что на начальном этапе "мозговой атаки" некоторые опорные точки зрения не были идентифицированы. Например, для сервисов "Удаленное обновление ПО" и "Удаленная диагностика" (см. рис. 6.4) необходимо иметь точку зрения об обслуживании программного обеспечения.

Рис. 6.4. Диаграмма идентификации точек зрения

На рис. 6.5 показано распределение сервисов для некоторых идентифицированных на рис. 6.4 точек зрения. Один и тот же сервис может быть соотнесен с несколькими точками зрения.

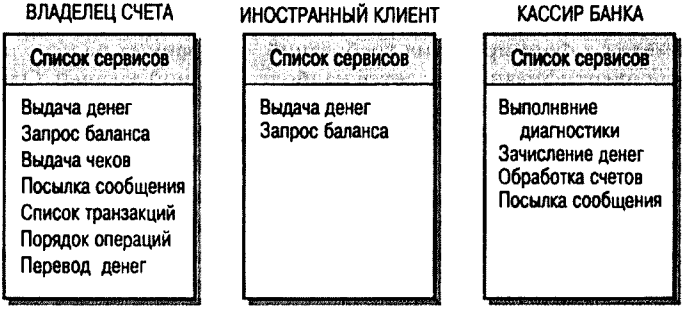


Рис. 6.5. Сервисы, соотнесенные с точками зрения

Точки зрения также определяют входные данные и управляющую информацию для сервисов. Например, банкомат должен определять остаток денег после выдачи наличных. На ранних этапах процесса формирования требований эти данные и управляющая информация идентифицируются просто по имени. На рис. 6.6 представлена управляющая информация для точки зрения владельца счета, сервисы которой показаны на рис. 6.5. Управляющая информация вводится с помощью кнопок на панели банкомата, данные – посредством клиентской карточки или клавиатуры банкомата.

ВЛАДЕЛЕЦ СЧЕТА	Управляющие данные	Входные данные
	Начало транзакции Отмена транзакции Конец транзакции Выбор сервиса	Данные на карточке PIN-код Требуемая сумма Сообщение

Рис. 6.6. Данные и управляющая информация

Информация, извлеченная из точек зрения, используется для заполнения форм шаблонов точек зрения и организации точек зрения в иерархию наследования. Это позволяет увидеть общие точки зрения и повторно использовать информацию в иерархии наследования. Сервисы, данные и управляющая информация наследуются подмножеством точек зрения. На рис. 6.7 показана часть иерархии точек зрения для системы банкоматов. Для простоты здесь представлены только сервисы, связанные с двумя точками зрения, без учета ряда точек зрения работников банка.

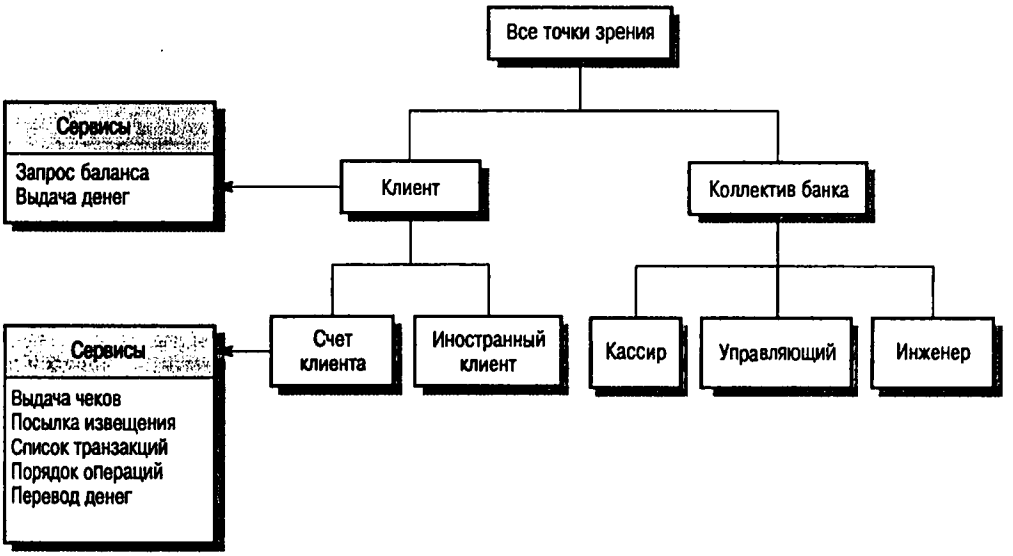


Рис. 6.7. Иерархия точек зрения

Следующая стадия процесса формирования требований – получение более детальной информации относительно сервисов, используемых сервисами данных, и управляющих данных. Эта информация извлекается из мнений лиц, формирующих требования, связанные с каждой опорной точкой зрения. Для этого используются шаблоны точек зрения и описания сервисов в виде сценариев событий. Эти сценарии обсуждаются в следующем разделе. Шаблоны точек зрения, описания сервисов и сценарии событий разрабатываются для всех идентифицированных опорных точек зрения и сервисов.

Поскольку в методе VORD необходимо обрабатывать большие объемы информации, поддержка CASE-средств значительно облегчает использование этого метода. CASE-средства для метода VORD можно свободно загрузить с Web-страницы данной книги.

6.2.2. Сценарии

Люди обычно легче воспринимают примеры из реальной жизни, чем абстрактные описания. Они легко понимают и могут оценить сценарии взаимодействия с программной системой. Специалисты могут использовать информацию, полученную из обсуждения сценариев взаимодействия с системой, для формулирования требований.

Сценарии особенно полезны для детализации уже сформулированных требований, поскольку описывают последовательность интерактивной работы пользователя с системой. Каждый сценарий описывает одно или несколько возможных взаимодействий. В настоящее время разработаны многочисленные формы сценариев, которые предоставляют различную информацию на разных уровнях детализации системы.

Сценарий начинается с общего описания, затем постепенно детализируется для создания полного описания взаимодействия пользователя с системой. В большинстве случаев сценарий включает следующее.

1. Описание состояния системы в начале сценария.
2. Описание нормального протекания событий.
3. Описание исключительных ситуаций и способов их обработки.
4. Информацию относительно других действий, которые можно осуществлять во время выполнения сценария.
5. Описание состояния системы после завершения сценария.

Первоначальное описание сценария может быть выполнено неформально в процессе опроса лиц, формирующих требования. Альтернативой может служить структурный подход – разработка сценариев событий или вариантов использования.

Сценарии событий

Сценарии событий используются в методе VORD для документирования поведения системы, представленного определенными событиями. Каждое событие, например вставку карточки в банкомат или выбор сервиса, можно документально подтвердить отдельным сценарием. Сценарии включают описание потоков данных, системных операций и исключительных ситуаций, которые могут возникнуть. На рис. 6.8 показан сценарий события "Начало транзакции", которое инициируется клиентом, вставляющим свою карточку в банкомат.

В схемах сценариев событий используется ряд условных обозначений.

1. Данные, поступающие в систему или исходящие из нее, представлены в эллипсах.
2. Управляющая информация показана стрелками в верхней части прямоугольников.

3. Внутрисистемные данные показаны справа от прямоугольников.
4. Исключительные ситуации показаны в нижней части прямоугольников. Там, где возможно несколько исключительных ситуаций, они заключаются в общий прямоугольник, как показано на рис. 6.8.
5. Имя следующего события, ожидаемого после завершения сценария, приводится в затенением прямоугольнике.

На рис. 6.8 видно, что, когда карточка вставлена, запрашивается персональный идентификационный номер клиента (PIN-код). Если карточка действительна, она может обрабатываться банкоматом, тогда управление переходит к следующей стадии сценария.

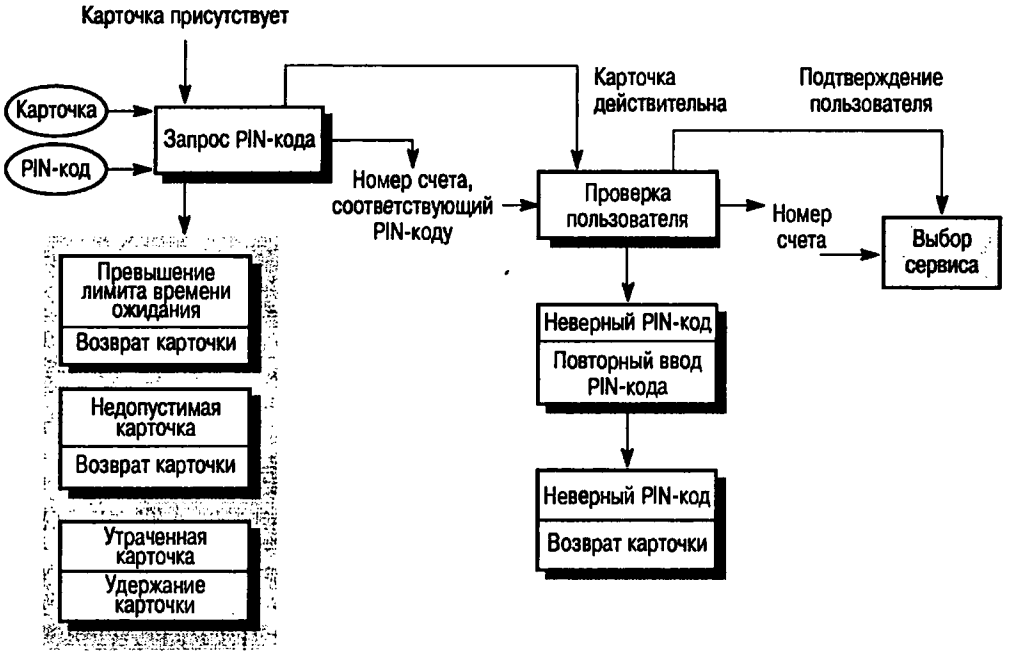


Рис. 6.8. Сценарий события "Начало транзакции"

На первой стадии сценария возможны три исключительные ситуации.

1. *Превышение лимита времени ожидания.* Клиент может не успеть ввести PIN-код в отведенное для ввода время. Карточка возвращается.
2. *Недопустимая карточка.* Карточка не опознается и возвращается.
3. *Удержание карточки.* Карточка удерживается банкоматом.

Каждую исключительную ситуацию можно определить более подробно, построив отдельные диаграммы потоков данных и управления. Общая диаграмма сценария также может быть снабжена комментариями с дополнительной информацией, содержащей описание действий, которые должны быть предприняты при возникновении исключительной ситуации.

"Проверка пользователя" — стадия проверки соответствия PIN-кода номеру счета клиента. Номер счета является выходными данными этой стадии. Возможная исключитель-

ная ситуация – ввод неверного PIN-кода, в этом случае PIN-код запрашивается снова. На диаграмме показано, что повторный запрос может опять привести к исключительной ситуации. Если повторно введенный PIN-код снова неверный, карточка возвращается. После события "Подтверждение пользователя" можно переходить к следующей стадии сценария "Выбор сервиса".

Варианты использования

Варианты использования (use-case) [186] – это методика формирования требований, основанная на сценариях. Они стали основой нотаций в языке моделирования UML при описании объектных моделей систем. В самой простой форме в варианте использования определены действующие лица (в UML они называются *актерами*), т.е. пользователи, вовлеченные во взаимодействие, и имена типов взаимодействия. На рис. 6.9 представлен простейший вариант использования, где показаны основные элементы обозначений. Действующие лица (актеры) изображены в виде стилизованных фигурок людей, а каждый класс взаимодействия представлен поименованным эллипсом. Множество вариантов использования охватывает все возможные взаимодействия, которые будут отражены в системных требованиях. На рис. 6.10 показаны варианты использования, описывающие взаимодействие читателей и библиотеки.

Рис. 6.9. Простейший вариант использования

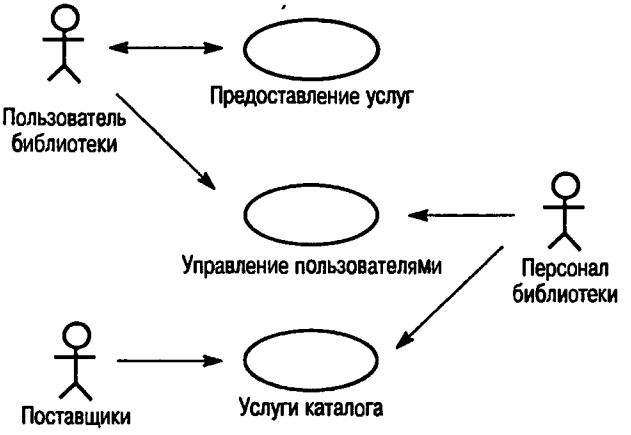


Рис. 6.10. Варианты использования для библиотеки

Иногда неясно, является ли вариант использования сценарием или, как предложено в [117], совокупностью сценариев, где каждый сценарий является "нитью", проходящей через вариант использования. В последнем случае возможны сценарии для нормального взаимодействия плюс сценарии для каждого возможного исключительного случая.

В языке UML предусмотрено совместное использование диаграмм последовательностей и вариантов использования, что существенно расширяет информационные возможности графического представления вариантов использования. Эти диаграммы показыва-

ют актеров, вовлеченных во взаимодействие, системные объекты, с которыми они взаимодействуют, и действия, которые связаны с этими объектами. На рис. 6.11 показаны взаимодействия при покупке книг и создании каталога библиотеки.

Язык моделирования UML фактически является *стандартом* для объектно-ориентированного представления вариантов использования, применяемых при формировании требований. Я покажу другие аспекты UML в главе 7, описывающей моделирование систем, и в главе 12, посвященной объектно-ориентированному проектированию.

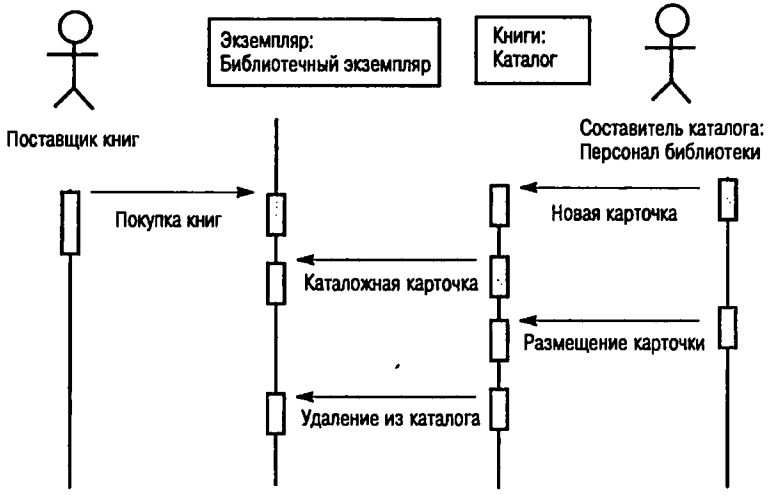


Рис. 6.11. Схема последовательности управления каталогом

6.2.3. Этнографический подход

Системы программного обеспечения не существуют изолированно от окружающего мира. Они эксплуатируются в определенной социальной и организационной среде, поэтому системные требования должны разрабатываться с учетом этой среды. Учет социальных и организационных требований часто имеет большое значение для успеха системы на рынке программных продуктов. Многие представленные на рынке программные системы практически не имеют спроса именно потому, что не учитывают важности социальных и организационных требований.

Этнографический подход к формированию системных требований используется для понимания и формирования социальных и организационных аспектов эксплуатации системы. Разработчик требований погружается в рабочую среду, где будет использоваться система. Его ежедневная работа связана с наблюдением и протоколированием реальных действий, выполняемых пользователями системы. Значение этнографического подхода заключается в том, что он помогает обнаружить неявные требования к системе, которые отражают реальные аспекты ее эксплуатации, а не формальные умозрительные процессы.

Обычно людям трудно четко описать все аспекты выполняемой работы, поскольку способ выполнения часто определяется их характером и практическим опытом. Они понимают свою работу, но не могут объяснить ее взаимосвязь с другими видами работ, выполняемых в организации. Социальные и организационные факторы, которые оказывают влияние на работу, но не являются очевидными, могут стать явными, если описаны беспристрастными наблюдателями.

В [328] этнографический подход использован для изучения работы офиса. Показано, что фактическая работа, выполняемая в офисе, более сложна и динамична, чем предусматривает простая модель, принятая для системы автоматизации делопроизводства. Это расхождение между реальной работой и моделью послужило причиной того, что офисные системы автоматизации не показывали той эффективности, на которую были рассчитаны.

Этнографический подход также применялся при формировании требований для систем управления воздушными перевозками [36, 168], систем управления диспетчерскими службами метрополитена [159], финансовых систем [169] и других [287, 65].

Этнографический подход к формированию требований можно объединить с прототипированием (рис. 6.12). Этнографический подход позволяет получить требования, которые учитываются в разрабатываемом прототипе. Кроме того, этнографический подход используется при решении конкретных проблем прототипирования и при оценивании созданного прототипа [322].

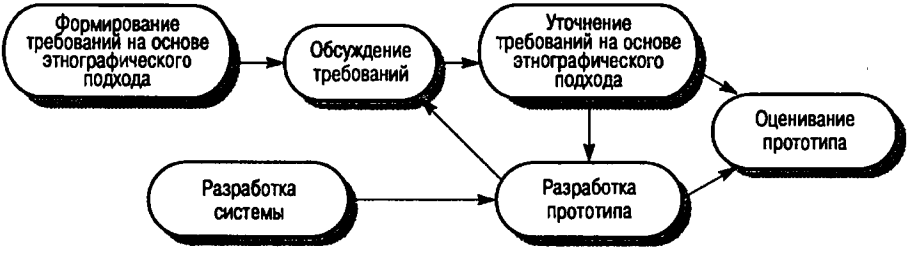


Рис. 6.12. Использование этнографического подхода и прототипирования для формирования требований

Этнографический подход позволяет детализировать требования для критических систем, чего не всегда можно добиться другими методами разработки требований. Однако, поскольку этот метод ориентирован на конечного пользователя, он не может охватить все требования предметной области и требования организационного характера. Поэтому он не является всеохватывающим подходом в формировании требований и должен использоваться совместно с такими подходами, как анализ вариантов использования.

6.3. Аттестация требований

Аттестация должна продемонстрировать, что требования действительно определяют ту систему, которую хочет иметь заказчик. Проверка требований важна, так как ошибки в спецификации требований могут привести к переделке системы и большим затратам, если будут обнаружены во время процесса разработки системы или после введения ее в эксплуатацию. Стоимость внесения в систему изменений, необходимых для устранения ошибок в требованиях, намного выше, чем исправление ошибок проектирования или кодирования. Причина в том, что изменение требований обычно влечет за собой значительные изменения в системе, после внесения которых она должна пройти повторное тестирование.

Во время процесса аттестации должны быть выполнены различные типы проверок документации требований.

1. *Проверка правильности требований.* Пользователь может считать, что система необходима для выполнения некоторых определенных функций. Однако дальнейшие размышления и анализ могут привести к необходимости введения дополнительных или новых функций. Системы предназначены для разных пользователей с различ-

ными потребностями, и поэтому набор требований будет представлять собой некоторый компромисс между требованиями пользователей системы.

2. *Проверка на непротиворечивость.* Спецификация требований не должна содержать противоречий. Это означает, что в требованиях не должно быть противоречащих друг другу ограничений или различных описаний одной и той же системной функции.
3. *Проверка на полноту.* Спецификация требований должна содержать требования, которые определяют все системные функции и ограничения, налагаемые на систему.
4. *Проверка на выполнимость.* На основе знания существующих технологий требования должны быть проверены на возможность их реального выполнения. Здесь также проверяются возможности финансирования и график разработки системы.

Существует ряд методов аттестации требований, которые можно использовать совместно или каждый в отдельности.

1. *Обзор требований.* Требования системно анализируются рецензентами. Этот процесс обсуждается в следующем разделе.
2. *Прототипирование.* На этом этапе прототип системы демонстрируется конечным пользователям и заказчику. Они могут экспериментировать с этим прототипом, чтобы убедиться, что он отвечает их потребностям. Методы прототипирования описаны в главе 8.
3. *Генерация тестовых сценариев.* В идеале требования должны быть такими, чтобы их реализацию можно было протестировать. Если тесты для требований разрабатываются как часть процесса аттестации, то часто это позволяет обнаружить проблемы в спецификации. Если такие тесты сложно или невозможно разработать, то обычно это означает, что требования трудно выполнить и поэтому необходимо их пересмотреть.
4. *Автоматизированный анализ непротиворечивости.* Если требования представлены в виде структурных или формальных системных моделей, можно использовать инструментальные CASE-средства для проверки непротиворечивости моделей. Этот процесс показан на рис. 6.13. Для автоматизированной проверки непротиворечивости необходимо построить базу данных требований и затем проверить все требования в этой базе данных. Анализатор требований готовит отчет обо всех обнаруженных противоречиях.

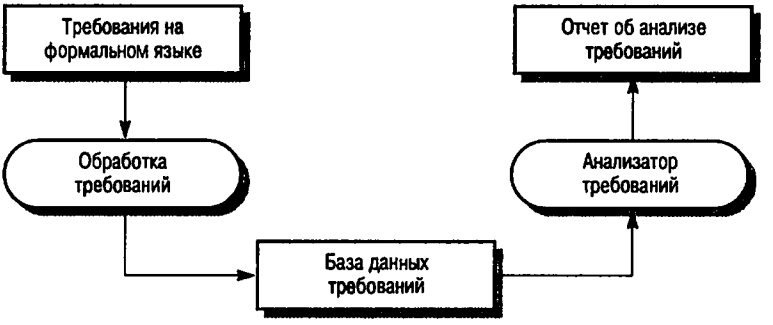


Рис. 6.13. Автоматизированный анализ непротиворечивости требований

Трудности аттестации требований нельзя недооценивать. Продемонстрировать, что все требования отвечают потребностям пользователя, очень трудно. Пользователи должны представить систему в действии и вообразить, как эта система впишется в их работу. Это трудно представить даже квалифицированным специалистам, не говоря уже о пользователях системы. В результате аттестации требований редко обнаруживаются все проблемы системной спецификации, поэтому в ней неизбежны изменения даже после согласования документа требований.

6.3.1. Обзор требований

Обзор требований — это процесс просмотра системной спецификации для нахождения неточных описаний и ошибок. К этому процессу привлекается большое количество лиц как со стороны заказчика, так и со стороны разработчиков. Обзор требований можно организовать так же, как и инспекцию программ (см. главу 19).

Обзор требований может быть неформальным и формальным. Неформальный обзор — это простое обсуждение требований с большим количеством лиц, участвующих в их формировании. Удивляет то, что часто связь между разработчиками системы и этими лицами заканчивается после формирования требований без документального подтверждения, что эти требования описывают ту систему, которая необходима данным лицам. Многие проблемы перед переходом к формальному обзору могут быть обнаружены простым обсуждением разрабатываемой системы с лицами, формирующими требования.

При формальном обзоре группа разработчиков должна "вести" заказчика через спецификацию, объясняя причину включения каждого требования. При этом проверяется непротиворечивость требований и их полнота.

Обнаруженные во время обзора противоречия, ошибки и упущения в требованиях должны быть зафиксированы документально. Затем эти документы передаются заказчику и разработчикам системы для принятия соответствующих мер.

6.4. Управление требованиями

Требования к большим системам ПО неизбежно будут изменяться в процессе их разработки. Причины этого многочисленны и разнообразны. Одной из причин является то, что во время процесса создания ПО понимание разработчиками поставленных перед ними задач будет неизбежно меняться, что вызывает необходимость возвращения к требованиям.

Кроме того, для больших программных систем, которые приходят на смену действующим, должна быть обеспечена преемственность. Хотя проблемы в работе со старой системой известны, трудно предсказать, какой эффект "улучшенная" система даст для организации. Если конечные пользователи имеют опыт работы с подобной системой, новые требования появляются по ряду причин.

1. Большие системы обычно имеют многообразный контингент пользователей. Разные пользователи имеют различные требования и приоритеты, которые могут быть противоречивыми или несовместимыми. Окончательный вариант системных требований представляет неизбежный компромисс между ними, который часто принимается только на заключительном этапе разработки системы.
2. Заказчики системы и ее пользователи — редко одни и те же люди. Заказчики формулируют требования, руководствуясь своими организационными и бюджетными ограничениями. Они могут входить в противоречие с требованиями конечных пользователей.

3. Деловая среда и техническое окружение системы изменяются, что должно найти отражение в системе. Например, может быть закуплено новое оборудование, может появиться необходимость сопряжения системы с другими системами, деловые приоритеты организации могут измениться, будут введены новые законодательство и стандарты и т.д. Изменения в аппаратных средствах особенно затрагивают нефункциональные системные требования.

Управление требованиями — это процесс управления изменениями системных требований. Процесс управления требованиями выполняется совместно с другими процессами разработки требований. Начало этого процесса планируется на то же время, когда начинается процесс первоначального формирования требований, непосредственно процесс управления требованиями должен начаться сразу после того, как черновая версия спецификации требований будет готова.

Описание процесса управления требованиями приведено ниже. Но прежде следует обсудить, почему требования неизбежно меняются, и объяснить, почему одни типы требований более подвержены изменениям, чем другие.

6.4.1. Постоянные и изменяемые требования

При формировании требований основное внимание сосредоточено на возможностях создаваемого ПО, бизнес-целях и других бизнес-системах организации. После формирования требований достигается более глубокое понимание потребностей пользователей, вследствие чего может возникнуть необходимость в изменении ранее сформулированных требований. Измененные требования отсылаются заказчику с объяснением причины сделанных изменений (рис. 6.14). Создание большой системы может занять несколько лет. За это время окружение и бизнес-требования к системе, несомненно, изменятся, что также должно найти отражение в измененных требованиях.

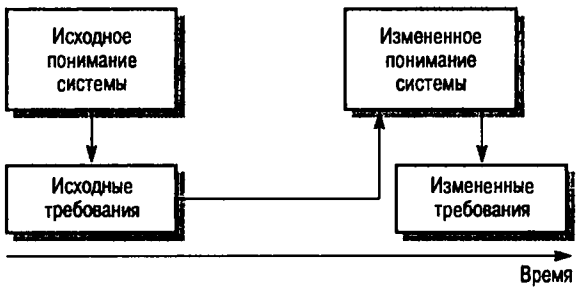


Рис. 6.14. Эволюция требований

С точки зрения разработки требования можно разделить на два класса.

1. *Постоянные требования.* Это относительно стабильные требования, которые исходят из основной деятельности организации и касаются непосредственно предметной области, где будет эксплуатироваться система.
2. *Изменяемые требования.* Эти требования отображают изменения, сделанные во время разработки системы или после ввода ее в эксплуатацию.

В работе [157] предложено классифицировать изменяемые требования по пяти классам. Но я считаю, что из этих классов два тесно связаны, поэтому предлагаю классификацию, показанную в табл. 6.1.

Таблица 6.1. Классификация изменяемых требований

Тип требований	Описание
Непостоянные требования	Требования, которые изменяются из-за изменений в окружении системы
Неожиданно возникающие требования	Требования, которые появляются во время разработки системы. В процессе проектирования может возникнуть необходимость добавления новых требований
Непрямые требования	Требования, которые являются результатом внедрения компьютерной системы, способной изменить организационные процессы и показать новые способы работы, которые приведут к новым системным требованиям
Вторичные требования	Требования, которые зависят от особенностей данной системы или от бизнес-проблем организации

6.4.2. Планирование управления требованиями

Планирование является первым этапом процесса управления требованиями. Управление требованиями очень дорого, и для каждого проекта на стадии планирования устанавливается необходимый уровень детализации управления требованиями. В процессе управления нужно отслеживать ряд вопросов, касающихся разработки требований.

1. *Идентификация требований.* Каждое требование должно быть однозначно определено, поскольку оно может пересекаться с другими требованиями. Пересечение требований можно обнаружить с помощью оперативного контроля.
2. *Управление процессом внесения изменений.* Это ряд операций, которые оценивают воздействие на систему вносимых изменений, а также стоимость изменений. Более подробно этот вопрос рассматривается в следующем разделе.
3. *Стратегия оперативного контроля.* Определяет отношения между требованиями, а также между требованиями и проектированием системы.
4. *Поддержка CASE-средств.* Управление требованиями предполагает обработку большого объема информации о требованиях. В этом процессе могут использоваться разнообразные инструментальные средства, например электронные таблицы или простые системы баз данных.

Существует много взаимозависимостей между требованиями, а также между требованиями и структурой системы. Кроме того, существует связь между требованиями и причинами, по которым эти требования были предложены. Если необходимо внести в требования изменения, в процессе управления нужно проследить влияние этих изменений на другие требования и саму систему. Оперативный контроль позволяет обнаружить связанные требования и отследить влияние одних требований на другие.

Существует три типа информации, используемой в оперативном контроле.

1. *Информация об источнике требования* связывает требование с лицами, которые предложили эти требования, и с логическим обоснованием этих требований. Если предложено изменение в требованиях, эта информация используется для определения лиц, которые могут обосновать эти изменения.
2. *Информация о требованиях* связывает требования внутри спецификации. Эта информация используется для оценки количества требований, которые затрагивают предложенные изменения.
3. *Информация о структуре системы* связывает требования с системными модулями, которые реализуют требования. Эта информация используется для оценки влияния предложенных изменений на систему и ее реализацию.

Информация для оперативного контроля часто представляется в виде специальных матриц, которые связывают требования с лицами, предложившими эти условия, требования между собой и требования с системными модулями. Если матрица оперативного контроля связывает требования между собой, то каждое требование представлено в матрице как строкой, так и столбцом. Тогда, если между требованиями существует зависимость, это указывается в ячейках на пересечении строк и столбцов, соответствующих этим требованиям. Пример простой матрицы зависимостей между требованиями показан в табл. 6.2. Символ U (от use – использование) на пересечении строки и столбца показывает, что требование в строке использует средства, определенные в требованиях, представленных в столбце. Символ R (от relation – связь, зависимость) означает, что существует некоторая взаимосвязь между требованиями. Например, оба требования определяют один и тот же системный модуль.

Таблица 6.2. Матрица оперативного контроля

Требования	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		U	R			U		U
1.2			U			R		
1.3	R			R				U
2.1			R		U			U
2.2								U
2.3		R		U				
3.1								R
3.2							R	

Матрицы оперативного контроля используются для управления небольшим набором требований, они стаиовятся громоздкими и неудобными для больших систем со многими требованиями. Для таких систем информацию оперативного контроля рациональнее держать в базе данных требований, где каждое требование связано явным образом с дру-

гими требованиями. Влияние изменений в требованиях можно проследить, используя средства просмотра базы данных.

Управление требованиями нуждается в автоматизированной поддержке, для чего можно использовать разнообразные CASE-средства. Средства поддержки необходимы для выполнения следующих операций.

1. *Хранение требований.* Информация о требованиях должна быть защищена, процесс хранения должен быть управляем, требования должны быть доступны для каждого участника процесса разработки требований.
2. *Управление изменениями требований.* Процесс управления изменениями (рис. 6.15) упрощается, если есть эффективные средства поддержки.
3. *Управление оперативным контролем.* Как отмечалось выше, средства поддержки оперативного контроля позволяют обнаруживать взаимосвязанные требования.

Для небольших программных систем нет необходимости использовать специализированные средства управления требованиями. Здесь для поддержки процесса управления можно обратиться к текстовым процессорам, электронным таблицам и обычным компьютерным базам данных. Однако для больших систем требуются специализированные средства поддержки. Информацию о средствах управления такими требованиями, как DOORS и Requisite Pro, можно найти на Web-странице данной книги.

6.4.3. Управление изменениями требований

Управление изменениями требований (см. рис. 6.15) должно применяться ко всем предложенным изменениям требований. Преимущество использования формального процесса управления изменениями состоит в том, что все предложенные изменения обрабатываются последовательно, при этом можно управлять и отслеживать внесение изменений в системную спецификацию.

Рис. 6.15. Управление изменениями требований

Процесс управления изменениями состоит из трех основных этапов.

1. *Анализ проблем изменения спецификации.* Процесс начинается с определения проблем в требованиях или с прямого предложения внесения изменений. На этой стадии проблема или предложенные изменения анализируются для проверки их обоснованности. Затем могут быть сделаны более определенные предложения относительно изменений в требованиях.
2. *Анализ изменений и расчет их стоимости.* Эффект от внесения предложенного изменения оценивается с использованием оперативного контроля. Стоимость изменений оценивается двумя показателями: стоимостью внесения изменения в спецификацию и стоимостью внесения изменений в структуру системы и непосредственно в программный код. По окончании этого этапа принимается решение, продолжать или нет внесение изменений в систему.

3. *Реализация изменений.* Реализация изменений в системной спецификации, структуре системы и программном коде.

Если требуется срочно внести изменения в требования, всегда существует соблазн сделать сначала изменения в системе, а затем задним числом изменить спецификацию. Это почти неизбежно приводит к тому, что готовая система не будет согласована с требованиями.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Процесс разработки требований включает анализ осуществимости создания системы, формирование и анализ требований, спецификирование требований, аттестацию требований и управление требованиями.
- Анализ требований является итерационным процессом, который включает анализ предметной области, сбор требований, их классификацию, разрешение противоречий, назначение приоритетов и проверку требований.
- Лица, участвующие в формировании требований, имеют различные приоритеты при формулировании требований. Поэтому сложные системы ПО необходимо анализировать с различных точек зрения.
- Социальные и организационные факторы оказывают большое влияние на системные требования.
- Аттестация требований — процесс проверки требований на достоверность, непротиворечивость, полноту и выполнимость. Обзоры требований и прототипирование являются основными методами аттестации требований.
- Процесс управления требованиями включает планирование управления, где определяется стратегия и процедуры управления требованиями, и непосредственно управление изменениями, где осуществляется анализ изменений и контроль за их реализацией.

Упражнения

6.1. Предложите, кто бы мог участвовать в формировании требований для университетской системы регистрации студентов. Объясните, почему почти неизбежно, что требования, сформулированные разными лицами, будут противоречивы.

6.2. Разрабатывается система ПО для автоматизации библиотечного каталога. Эта система будет содержать информацию относительно всех книг в библиотеке и будет полезна библиотечному персоналу, абонентам и читателям. Система должна иметь средства просмотра каталога, средства создания запросов и средства, позволяющие пользователям резервировать книги, находящиеся в данный момент на руках. Определите основные опорные точки зрения, которые необходимо учесть в спецификации системы, и покажите их взаимоотношения, используя диаграмму иерархии точек зрения.

6.3. Для трех точек зрения, определенных в системе библиотечного каталога, укажите сервисы и соответствующие данные, которые обеспечиваются этими точками зрения, и события, которые управляют этими сервисами.

6.4. Для сервисов, определенных в упражнении 6.3, укажите наиболее важные нефункциональные ограничения.

6.5. Приведите пример типа системы, где социальные и политические факторы могут иметь сильное влияние на системные требования.

6.6. Кто должен проводить обзор требований? Нарисуйте модель процесса обзора требований.

6.7. Почему матрицы оперативного контроля трудны для управления в случае большого количества системных требований? Предложите механизм структурирования требований, основанный на точках зрения, который может помочь решить эту проблему.

6.8. Существуют ситуации, когда изменения в системе вносятся прежде, чем изменения в требованиях будут одобрены. Предложите модель процесса внесения срочных изменений в систему, который гарантирует согласованность системы и спецификации требований.

6.9. Ваша компания использует стандартный метод анализа требований. В процессе работы вы обнаружили, что этот метод не учитывает социальные факторы, важные для системы, которую вы анализируете. Ваш руководитель дал вам ясно понять, какому методу анализа нужно следовать. Обсудите, что вы должны делать в такой ситуации.

7 Модели систем

Цели

Эта глава знакомит с различными типами моделей систем, которые используются в процессе разработки требований. Прочитав ее, вы должны:

- понимать, почему важны модели рабочего окружения систем;
- знать концепции моделирования поведения, данных и объектов систем;
- ознакомиться с нотациями, применяемыми в унифицированном языке моделирования UML, и с тем, как они используются при разработке различных типов моделей систем;
- знать инструментальные CASE-средства, используемые при моделировании систем.

Содержание

Пользовательские требования обычно пишутся на естественном языке, поскольку они должны быть понятны даже не специалистам в области разработки ПО. Однако более детализированные системные требования должны описываться более "техническим" способом. Одной из широко используемых методик документирования системных требований является построение ряда моделей системы. Эти модели используют графические представления, показывающие решение как исходной задачи, для которой создается система, так и разрабатываемой системы. Как правило, графические представления более понятны, чем детальное описание системных требований на естественном языке. Модели являются связующим звеном между процессом анализа исходной задачи и процессом проектирования системы.

Модели можно использовать в процессе анализа существующей системы, которую нужно или заменить, или модифицировать, а также для формирования системных требований. Модели могут представить систему в различных аспектах.

1. Внешнее представление, когда моделируется окружение или рабочая среда системы.
2. Описание поведения системы, когда моделируется ее поведение.
3. Описание структуры системы, когда моделируется системная архитектура или структуры данных, обрабатываемых системой.

Эти три типа представления систем раскрыты в данной главе; кроме того, здесь рассматривается объектное моделирование, которое до некоторой степени объединяет поведенческое и структурное моделирование.

Такие структурные методы, как структурный анализ систем [91, 12*, 24*] и объектно-ориентированный анализ [302, 54, 33*], обеспечивают основу для детального моделирования системы как части процесса постановки и анализа требований. Большинство структурных методов работают с определенными типами системных моделей. Эти методы обычно определяют процесс, который используются для построения моделей, и набор правил, которые применяются к этим моделям. Для поддержки структурных методов существуют различные CASE-средства (обсуждаемые в разделе 7.5), включающие редакторы моделей, автоматизированную систему документирования и инструменты проверки моделей.

Однако методы структурного анализа имеют ряд недостатков.

1. Они не обеспечивают эффективной поддержки формирования нефункциональных системных требований.
2. Обычно руководства по этим методам не содержат советов, которые помогали бы пользователям решить, подходит ли данный метод для решения конкретной задачи.
3. В результате применения этих методов часто получается объемная документация, при этом суть системных требований скрывается за массой несущественных деталей.
4. Построенные модели очень детализированы и трудны для понимания. Обычные пользователи не могут реально проверить действенность этих моделей.

Практическая разработка требований не должна ограничиваться только моделями, построенными с помощью тех или иных методов. Например, объектно-ориентированные методы обычно не предполагают применения для разработки моделей потоков данных. Однако, исходя из моего опыта, такие модели полезны для объектно-ориентированного анализа, поскольку отражают понимание системы конечным пользователем и могут помочь идентифицировать объекты и действия с ними.

Наиболее важным аспектом системного моделирования является то, что оно опускает детали. Модель является абстракцией системы и легче поддается анализу, чем любое дру-

гое представление этой системы. В идеале *представление* системы должно сохранять всю информацию относительно представляемого объекта. *Абстракция* является упрощением и определяется выбором наиболее важных характеристик системы.

Различные типы системных моделей основаны на разных подходах к абстракции. Например, модель потоков данных концентрирует внимание на прохождении данных через систему и на функциональных преобразованиях этих данных. Модель оставляет без внимания структуру данных. И наоборот, модель "сущность–связь" предполагает документирование системных данных и их взаимосвязь, не касаясь системных функций.

Приведем типы системных моделей, которые могут создаваться в процессе анализа систем.

1. *Модель обработки данных.* Диаграммы потоков данных показывают последовательность обработки данных в системе.
2. *Композиционная модель.* Диаграммы "сущность–связь" показывают, как системные сущности составляются из других сущностей.
3. *Архитектурная модель.* Эти модели показывают основные подсистемы, из которых строится система.
4. *Классификационная модель.* Диаграммы наследования классов показывают, какие объекты имеют общие характеристики.
5. *Модель "стимул–ответ".* Диаграммы изменения состояний показывают, как система реагирует на внутренние и внешние события.

Эти типы моделей описаны далее в главе. Везде, где возможно, я использую обозначения унифицированного языка моделирования UML, который является стандартом языка моделирования, особенно для объектно-ориентированного моделирования [55, 303, 5*, 30*]. В тех случаях, когда UML не предусматривает подходящих нотаций, для описания моделей я использую простые интуитивные обозначения.

7.1. Модели системного окружения

На ранних этапах формирования требований необходимо определить границы системы. Этот этап предполагает работу с лицами, участвующими в формировании требований (см. главу 6), для того чтобы разграничить систему и ее рабочее окружение. В некоторых случаях границы между системой и ее окружением относительно ясны. Например, когда новая система заменяет существующую, ее рабочее окружение обычно совпадает с окружением существующей системы.

В других случаях необходим дополнительный анализ. Например, рабочее окружение разрабатываемого набора CASE-средств может включать существующую базу данных, сервисы которой используются системой, но набор средств может также иметь внутреннюю базу данных. Если база данных уже существует, определение границ между ними может оказаться сложной технической и управленческой проблемой. Только после проведения дополнительного анализа можно будет принять решение о том, что является, а что не является частью разрабатываемой системы.

На определение системного окружения могут также влиять социальные и организационные ограничения, т.е. граница системы может определяться не только техническими факторами. Например, система может быть очерчена так, что при ее разработке не будет необходимости консультироваться с менеджерами; при другом определении границ может возрасти ее стоимость или возникнет необходимость расширить отдел разработки и т.п.

После определения границ между системой и ее окружением далее специфицируется само рабочее окружение и связи между ним и системой. Обычно на этом этапе строится простая структурная модель, подобная представленной на рис. 7.1 модели структуры окружения информационной системы, управляющей сетью банкоматов. Структурные модели высокого уровня обычно являются простыми блок-схемами, где каждая подсистема представлена именованным прямоугольником, а линии показывают, что существуют некоторые связи между подсистемами.

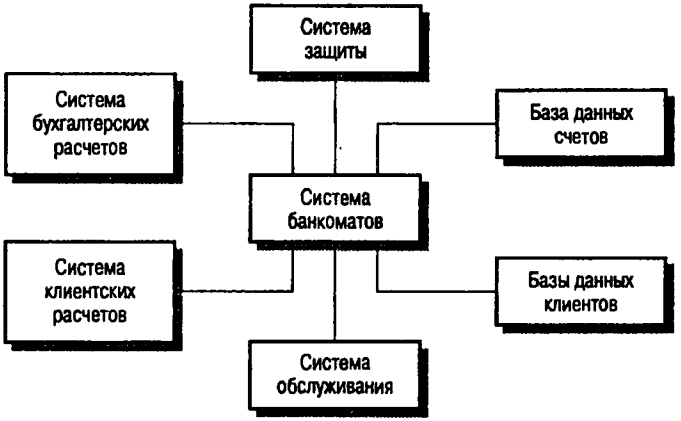


Рис. 7.1. Рабочее окружение системы управления банкоматами

На рис. 7.1 показано, что каждый банкомат присоединен к базе данных счетов, к локальной системе клиентских расчетов, к системе защиты и к системе обслуживания банкоматов. Система также соединена с базой данных клиентов, контролирующей сеть банкоматов, и с локальной бухгалтерской системой.

Структурные модели описывают непосредственное рабочее окружение системы. Но они не показывают связи между другими системами в окружающей среде, которые не соединены непосредственно с разрабатываемой системой, но могут на нее влиять. Например, внешние системы могут производить данные для системы или использовать данные, произведенные системой. При этом они могут быть соединены между собой и системой через сеть или не соединены вообще. Они могут физически соприкасаться или располагаться в разных зданиях. Все эти взаимоотношения могут влиять на требования к разрабатываемой системе и должны быть приняты во внимание.

Таким образом, простые структурные модели обычно дополняются моделями других типов, например моделями процессов, которые показывают взаимодействия в системе, или моделями потоков данных (описаны в следующем разделе), которые показывают последовательность обработки и перемещения данных внутри системы и между другими системами в окружающей среде.

На рис. 7.2 представлена модель процесса заказа оборудования организацией. Она включает определение необходимого оборудования, поиск и выбор поставщиков, заказ, поставку и проверку оборудования после поставки. Для определения системы компьютерной поддержки этого процесса необходимо решить, какие из этих действий будут выполняться системой, а какие окажутся внешними по отношению к ней. На рис. 7.2 пунктирной линией ограничены действия, выполняемые внутри такой системы.

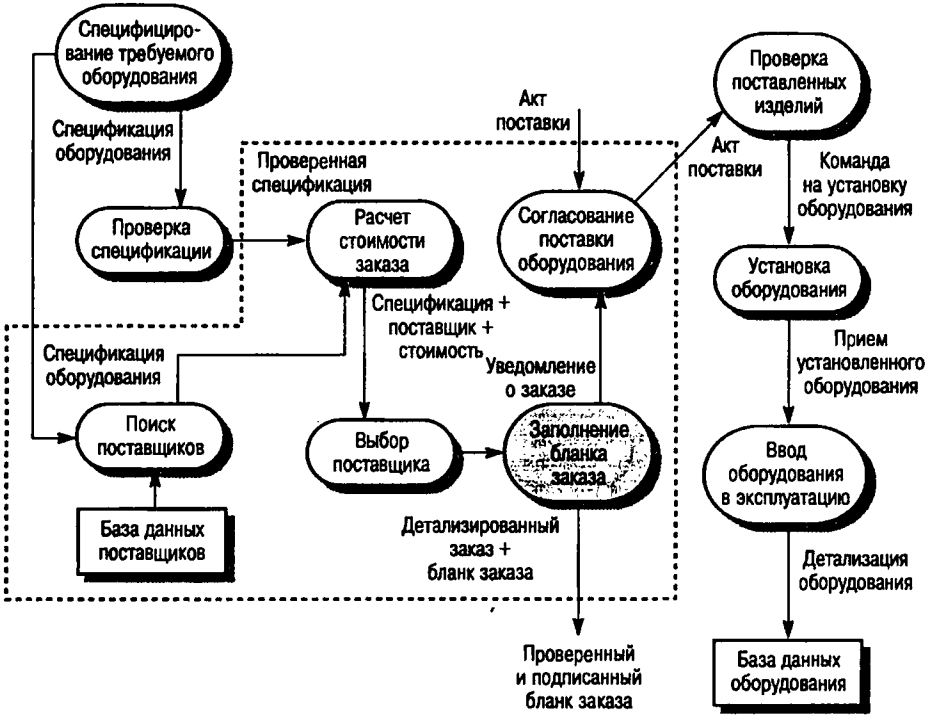


Рис. 7.2. Модель процесса приобретения оборудования

7.2. Поведенческие модели

Эти модели используются для описания общего поведения системы. Здесь рассматривается два типа поведенческих моделей – модель потоков данных, которая представляет обработку данных в системе, и модель конечного автомата, которая показывает реакцию системы на события. Эти модели можно использовать отдельно или совместно, в зависимости от типа разрабатываемой системы.

Большинство бизнес-систем прежде всего управляют данными. Они также управляют вводом данных в систему и сравнительно мало занимаются обработкой внешних событий. Для таких систем модель потоков данных может содержать все, что необходимо для описания поведения системы. В противоположность им системы реального времени управляют событиями с минимальной обработкой данных. Модель конечного автомата (обсуждаемая в разделе 7.2.2) является наиболее эффективным способом описания их поведение. Другие классы систем управляют как данными, так и событиями, поэтому для их представления необходимы оба типа моделей.

7.2.1. Модели потоков данных

Модели потока данных – это интуитивно понятный способ показа последовательности обработки данных внутри системы. Нотации, используемые в этих моделях, описывают обработку данных с помощью системных функций, а также хранение и перемещения данных между системными функциями. Модели потоков данных стали широко использовать-

ся после публикации книги о структурном системном анализе [91]. На базе этого фундаментального исследования было разработано множество методов анализа систем.

Модели потоков данных используются для показа последовательности шагов обработки данных. Эти шаги обработки или преобразования данных выполняются программными функциями. В сущности, диаграммы потоков данных используются для документирования программных функций перед проектированием системы. Анализ модели обработки данных может быть выполнен специалистами вручную или с помощью компьютера.

Модель потоков данных, показанная на рис. 7.3, представляет действия, выполняемые при оформлении заказа на оборудование. Это описание части процесса размещения заказа на оборудования, представленного на рис. 7.2. Данная модель показывает процесс перемещения бланка заказа при его обработке.

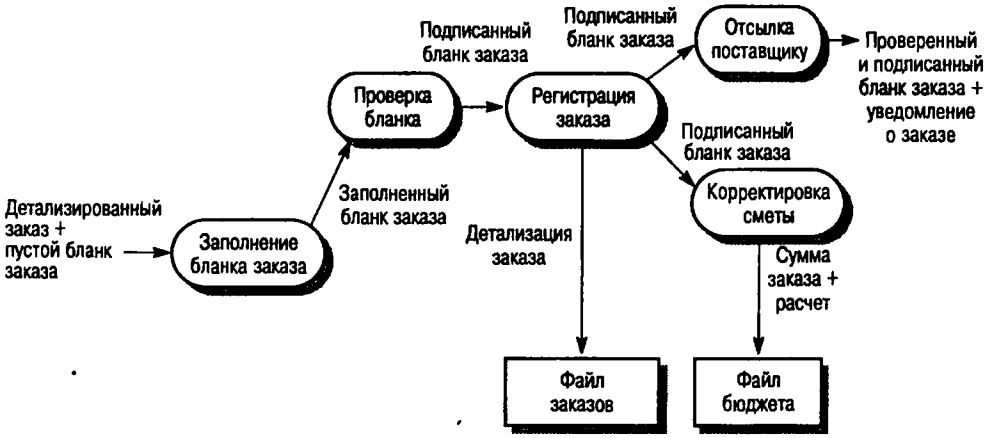


Рис. 7.3. Диаграмма потоков данных при обработке бланка заказа

В диаграммах потоков данных используются следующие обозначения (см. рис. 7.3): закругленные прямоугольники соответствуют этапам обработки данных; стрелки, снабженные примечаниями с названием данных, представляют потоки данных; прямоугольники соответствуют хранилищам или источникам данных.

Модели потоков данных ценны тем, что они прослеживают и документируют перемещение данных по системе, помогая тем самым аналитикам понять этот процесс. Преимущество диаграмм потоков данных в том, что они, в отличие от других моделей, просты и интуитивно понятны. Поэтому их можно объяснить потенциальным пользователям системы, которые затем могут участвовать в ее анализе.

Модели потоков данных показывают функциональную структуру системы, где каждое преобразование данных соответствует одной системной функции. Иногда модели потоков данных используют для описания потоков данных в рабочем окружении системы. Такая модель показывает, как различные системы и подсистемы обмениваются информацией. Подсистемы окружения не обязаны быть простыми функциями. Например, одна подсистема может быть сервером базы данных с довольно сложным интерфейсом. На рис. 7.4 показана подобная диаграмма потоков данных. В этом примере закругленные прямоугольники представляют подсистемы.

Рис. 7.4. Диаграмма потоков данных комплекса CASE-средств

7.2.2. Модели конечных автоматов

Модели конечных автоматов[1] используются для моделирования поведения системы, реагирующей на внутренние или внешние события. Такая модель показывает состояние системы и события, которые служат причиной перехода системы из одного состояния в другое. Модель не показывает поток данных внутри системы. Этот тип модели особенно полезен для моделирования систем реального времени, поскольку этими системами обычно управляют входные сигналы, приходящие из окружения системы. Например, система сигнализации, рассмотренная в главе 13, реагирует на сигналы датчиков перемещения и дверных датчиков и т.д.

Модели конечных автоматов являются неотъемлемой частью методов проектирования систем реального времени [155, 156, 338, 35*]. В работе [155] определены *диаграммы состояний*, которые стали основой системы нотаций в языке моделирования UML.

Модель конечного автомата системы предполагает, что в любое время система находится в одном из возможных состояний. При получении входного сигнала или стимула система может изменить свое состояние. Например, система, управляющая клапаном, при получении команды оператора (стимул) может перейти из состояния "Клапан открыт" к состоянию "Клапан закрыт".

На рис. 7.5 показана модель конечного автомата (диаграмма состояний) простой микроволновой печи, оборудованной кнопками включения питания, таймера и запуска системы.

Реальная микроволновая печь на самом деле намного сложнее описанной здесь системы. Вместе с тем эта модель показывает все основные средства системы. Для упрощения модели я предполагаю такую последовательность действий при использовании печи.

1. Выбор уровня мощности (половинная или полная).
2. Ввод времени работы печи.
3. Нажатие кнопки запуска, после чего печь работает заданное время.

Для безопасности печь не должна действовать при открытой двери, по окончании работы должен прозвучать звуковой сигнал. Печь имеет простой дисплей, на котором отображаются различные предупреждения и сообщения.

Нотация UML, которую я использую для описания модели конечного автомата и диаграммы состояний, разработана для моделирования поведения объектов. Но ее можно использовать для любого типа моделей конечного автомата. В этой нотации закругленные прямоугольники соответствуют состояниям системы. Они содержат краткое описание действий, выполняемых в этом состоянии. Помеченные стрелки представляют стимулы (или входные сигналы), которые приводят к переходу системы из одного состояния в другое.

---

[1] *В русской научной литературе модели конечных автоматов, в том контексте, как они представлены в книге, обычно просто называют диаграммами состояний, без упоминания "конечных автоматов".* – Прим. ред.

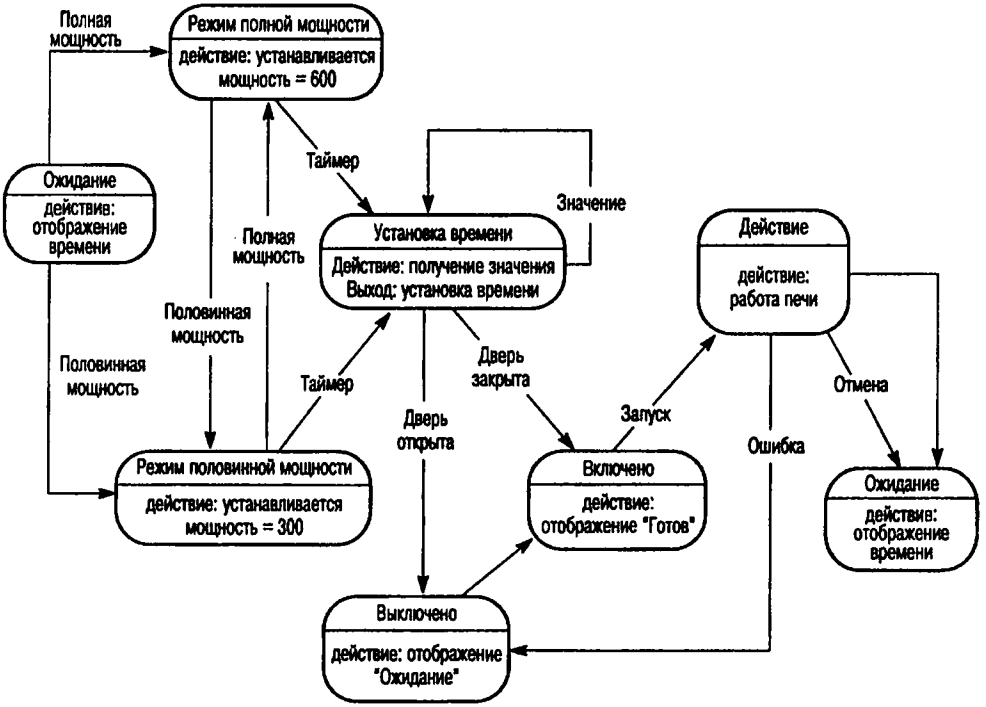


Рис. 7.5. Диаграмма состояний автомата микроволновой печи

На рис. 7.5 видно, что система первоначально реагирует на нажатие кнопок полной или половинной мощности. Пользователь может изменить свое решение после нажатия одной из этих кнопок и выбрать другую кнопку. Кнопка запуска сработает только после задания времени работы печи и закрытия двери. Нажатием кнопки запуска начинается работа печи в течение указанного времени.

Нотации UML позволяют определять действия, которые выполняются в том или ином состоянии. Но для создания системной спецификации необходима более детальная информация о стимулах и состояниях системы (табл. 7.1). Эта информация может храниться в словаре данных, как будет показано далее в этом разделе.

Таблица 7.1. Описание состояний и стимулов микроволновой печи

Состояние	Описание
Ожидание	Печь находится в состоянии ожидания входных данных. На дисплее высвечивается текущее время
Режим половинной мощности	Мощность печи устанавливается на 300 Вт. На дисплее отображается "Половинная мощность"
Режим полной мощности	Мощность печи устанавливается на 600 Вт. На дисплее отображается "Полная мощность"
Установка времени	Пользователем устанавливается время работы печи. Дисплей показывает заданное время
Выключено	В целях безопасности печь выключена. Внутренность печи освещена. Дисплей показывает "Не готов"

Окончание табл. 7.1

Состояние	Описание
Включено	Питание печи включено. Внутри печи света нет. Дисплей высвечивает "Готов"
Работа	Печь работает. Внутри печи включается свет. Дисплей отображает обратный отсчет таймера. По окончании работы звучит звуковой сигнал в течение 5 секунд. Свет включен. Пока звучит сигнал, дисплей высвечивает "Приготовление закончено"
Стимулы	Описание
Половинная мощность	Пользователь нажимает кнопку режима половинной мощности
Полная мощность	Пользователь нажимает кнопку режима полной мощности
Таймер	Пользователь нажимает одну из кнопок таймера
Число	Пользователь вводит число
Дверь открыта	Переключатель двери печи в состоянии "Не закрыто"
Дверь закрыта	Переключатель двери печи в положении "Закрыто"
Запуск	Пользователь нажимает кнопку запуска
Отмена	Пользователь нажимает кнопку отмены

Основная проблема метода конечного автомата состоит в том, что число возможных состояний может быть очень велико. Поэтому для моделей больших систем необходима структуризация возможных состояний системы. Один из способов структуризации состоит в использовании суперсостояний, которые объединяют ряд отдельных состояний. Такое суперсостояние подобно одному состоянию модели высокого уровня, которое детализируется на отдельной диаграмме. Для иллюстрации этого понятия рассмотрим состояние Работа модели микроволновой печи (см. рис. 7.5). Это суперсостояние более детально показано на рис. 7.6.

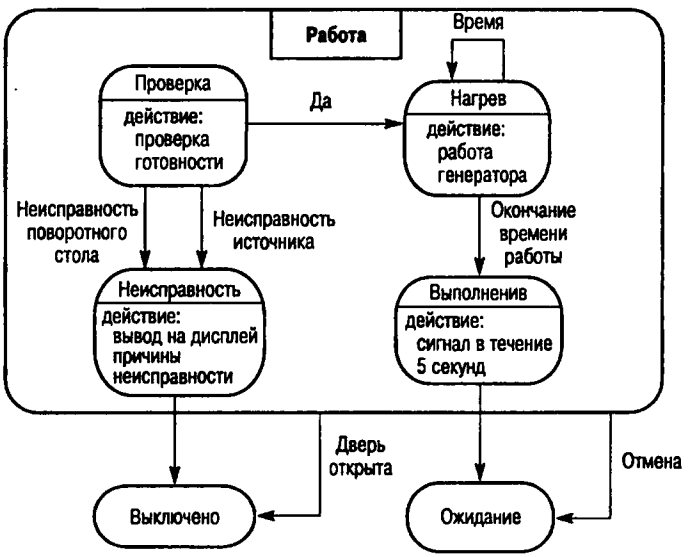


Рис. 7.6. Работа микроволновой печи

Состояние Работа включает ряд подсостояний. Диаграмма на рис. 7.6 показывает, что в состоянии Работа сначала проверяется готовность печи к работе и, если обнаружены какие-либо проблемы, включается аварийная сигнализация и печь выключается. В состоянии Нагрев работает микроволновой генератор в течение указанного времени, по завершении его работы автоматически подается звуковой сигнал. Если во время работы печи будет открыта дверь, система переходит в состояние Выключено, как показано на рис. 7.5.

7.3. Модели данных

Многие большие программные системы используют информационные базы данных. В одних случаях эта база данных существует независимо от программной системы, в других — специально создается для разрабатываемой системы. Важной частью моделирования систем является определение логической формы данных, обрабатываемых системой.

Наиболее широко используемой методологией моделирования данных является моделирование типа "сущность–связь–атрибут", которое показывает структуру данных, их атрибуты и отношения между ними[2]. Этот метод моделирования был предложен в середине 1970-х годов Ченом (Chen, [71]); с тех пор разработано несколько вариантов этого метода [76, 154, 170, 1*].

Язык моделирования UML не имеет определенных обозначений для этого типа моделей данных, что желательно для объектно-ориентированного процесса разработки ПО, где для описания систем используются объекты и их отношения. Если сущностям поставить в соответствие простейшие классы объектов (без ассоциированных методов), тогда в качестве моделей данных можно использовать модели классов UML совместно с именованными ассоциациями между классами. Хотя такие модели данных не могут служить примером "хорошего" языка моделирования, удобство использования стандартных обозначений UML перевешивает возможные несовершенства таких конструкций.

Для описания структуры обрабатываемой информации модели данных часто используются совместно с моделями потоков данных. На рис. 7.7 представлена модель данных для системы проектирования ПО. Такую систему можно реализовать на основе комплекса инструментальных CASE-средств, показанного на рис. 7.4.

Проекты структуры ПО представляются ориентированными графами. Они состоят из набора узлов различных типов, соединенных дугами, отображающими связи между структурными узлами. В системе проектирования присутствуют средства вывода на дисплей этого графа (т.е. структурной диаграммы) и его преобразования к виду, удобному для хранения в базе данных проектов. Система редактирования выполняет преобразования структурной диаграммы из формата базы данных в формат, позволяющий отобразить ее на экране монитора в виде блок-схемы. Информация, предоставляемая редактором другим средствам анализа проекта, должна включить логическое представление графа проекта. Конечно, эти средства "не интересуются" деталями физического представления растрового изображения графа. Они работают с объектами, их логическими атрибутами и связями между ними.

---

[2] *К сожалению, автор не приводит определения понятий "сущность", "связь" и "атрибут", хотя в рассматриваемых методах моделирования они являются базовыми и строго формализованы. Приведем их краткое описание.* Сущность *(entity)* – *реальный или абстрактный объект, имеющий определяющее значение для рассматриваемой системы (это может быть объект как самой системы, так и ее окружения). Каждая сущность должна иметь уникальное имя и обладать одним или несколькими атрибутами, которые либо принадлежат сущности, либо наследуются через связь. Сущность соответствует классу (или типу) объектов, а не конкретному экземпляру класса.* Связь *(relation)* – *поименованная связь (ассоциация) между двумя сущностями. Именование связей осуществляется с помощью глаголов (например,* имеет, определяет, принадлежит *и т.п.).* Атрибут *(attribute)* – *любая характеристика сущности (предназначается для идентификации, классификации, численной параметризации или описания состояния сущности). Значения атрибутов однозначно идентифицируют экземпляр сущности.* – Прим. ред.

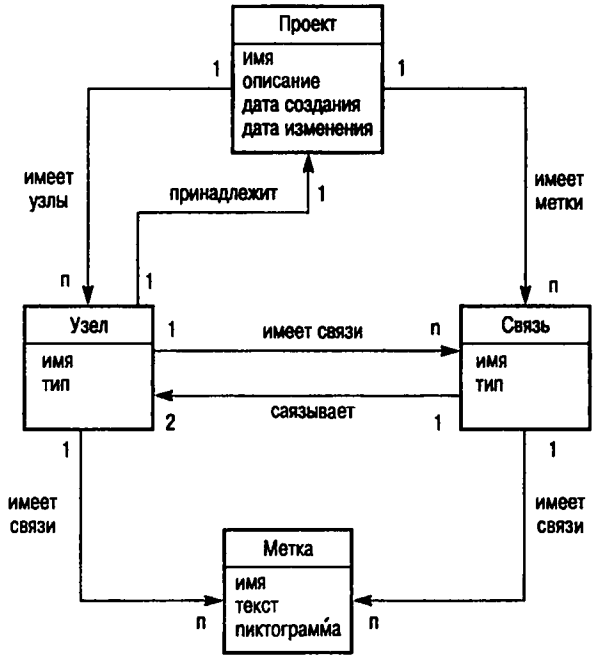


Рис. 7.7. Модель данных для системы проектирования ПО

На рис. 7.7 видно, что проект имеет атрибуты **имя**, **описание**, **дата создания** и **дата изменения**. Проект состоит из узлов и связей между ними (т.е. дуг). Узлы и связи имеют атрибуты **имя** и **тип**. Они могут иметь набор меток, которые хранят другую описательную информацию. Каждая метка имеет атрибуты **имя**, **пиктограмма** и **текст**.

Подобно всем графическим моделям, модели "сущность–связь–атрибут" недостаточно детализированы, поэтому они обычно дополняются более подробным описанием объектов, связей и атрибутов, включенных в модель. Эти описания собираются в словари данных или репозитории. Словари данных необходимы при разработке моделей системы и могут использоваться для управления информацией, содержащейся во всех моделях системы.

Упрощенно словарь данных — это просто алфавитный список имен, которые включены в различные модели системы. Вместе с именем словарь должен содержать описание именованного объекта, а если имя соответствует сложному объекту, может быть представлено описание построения этого объекта. Другая информация, например дата создания или фамилия разработчика, может приводиться в зависимости от типа разрабатываемой модели.

Перечислим преимущества использования словаря даиных.

1. Существует механизм управления именами. При разработке большой системной модели, вероятно, придется изобретать имена для сущностей н связей. Эти имена должны быть уникальными. Программа словаря данных может проверять уникальность имен и сообщать об их дублировании.
2. Словарь может служить хранилищем организационной информации, которая может связать анализ, проектирование, реализацию и модернизацию системы. Вся информация о системных объектах и сущностях находится в одном месте.

Все имена, используемые в системе (имена сущностей, типов, связей, атрибутов и системных сервисов), должны быть введены в словарь данных. Программные средства слова-

ря должны обеспечить создание новых записей, их хранение и запросы к словарю. Такое программное обеспечение может быть интегрировано с другими программными средствами. Большинство CASE-средств, которые применяются для моделирования систем, могут поддерживать словари данных.

В примере словаря данных, приведенном в табл. 7.2, представлены имена, взятые из модели данных системы проектирования (см. рис. 7.7). Это упрощенный пример, где опущены некоторые имена и сокращена соответствующая информация.

Таблица 7.2. Пример словаря данных

Имя	Описание	Тип	Дата
имеет метки	Отношение 1:N между сущностями типа Узел или Связь и сущностями типа Метка[3]	Связь	5.10.1998
Метка	Содержит информацию об узлах и связях. Метки представляются пиктограммами и соответствующим текстом	Сущность	8.12.1998
Связь	Отношение 1:1 между сущностями, представленными узлами. Связи имеют тип и имя	Связь	8.12.1998
имя (метка)	Каждая метка имеет имя, которое должно быть уникальным	Атрибут	8.12.1998
имя (узел)	Каждый узел имеет имя, которое должно быть уникальным. Имя может содержать до 64 символов	Атрибут	5.11.1998

7.4. Объектные модели

Объектно-ориентированный подход широко используется при разработке программного обеспечения, особенно для разработки интерактивных систем. В этом случае системные требования формируются на основе объектной модели, а программирование выполняется с помощью объектно-ориентированных языков, таких, как Java или C++.

Объектные модели, разработанные для формирования требований, могут использоваться как для представления данных, так и для процессов их обработки. В этом отношении они объединяют модели потоков данных и семантические модели данных. Они также полезны для классификации системных сущностей и могут представлять сущности, состоящие из других сущностей.

Для некоторых классов систем объектные модели — естественный способ отображения реально существующих объектов, которые находятся под управлением системы. Например, для систем, обрабатывающих информацию относительно конкретных объектов (таких, как автомобили, самолеты, книги и т.д.), которые имеют четко определенные атрибуты. Более абстрактные высокоуровневые сущности, например библиотеки, медицинские регистрирующие системы или текстовые редакторы, труднее моделировать в виде классов объектов, поскольку они имеют достаточно сложный интерфейс, состоящий из независимых атрибутов и методов.

Объектные модели, разработанные во время анализа требований, несомненно, упрощают переход к объектно-ориентированному проектированию и программированию. Однако конечные пользователи часто считают объектные модели неестественными и трудными для понимания. Часто они предпочитают функциональные представления процес-

[3] *"Отношение 1:N" – это сокращение от "отношение один-ко-многим". Аналогично "отношение 1:1" (см. далее в таблице) – "отношение один-к-одному".* – Прим. ред.

сов обработки данных. Поэтому полезно дополнить их моделями потоков данных, чтобы показать сквозную обработку данных в системе.

Класс объектов – это абстракция множества объектов, которые определяются общими атрибутами (как в семантической модели данных) и сервисами или операциями, которые обеспечиваются каждым объектом. Объекты – это исполняемые сущности с атрибутами и сервисами класса объектов. Объекты представляют собой реализацию класса, на основе одного класса можно создать много различных объектов. Обычно при разработке объектных моделей основное внимание сосредоточено на классах объектов и их отношениях.

Модели систем, разрабатываемые при формировании требований, должны отображать реальные сущности, принадлежащие классам объектов. Классы не должны содержать информацию об отдельных системных объектах. Можно разработать различные типы объектных моделей, показывающие, как классы связаны друг с другом, как объекты агрегируются из других объектов, как объекты взаимодействуют с другими объектами и т.д. Эти модели расширяют понимание разрабатываемой системы.

Идентификация объектов и классов объектов считается наиболее сложной задачей в процессе объектно-ориентированной разработки систем. Определение объектов – это основа для анализа и проектирования системы. Методы определения объектов описаны в главе 12. Здесь рассматриваются лишь некоторые объектные модели, полезные для анализа систем.

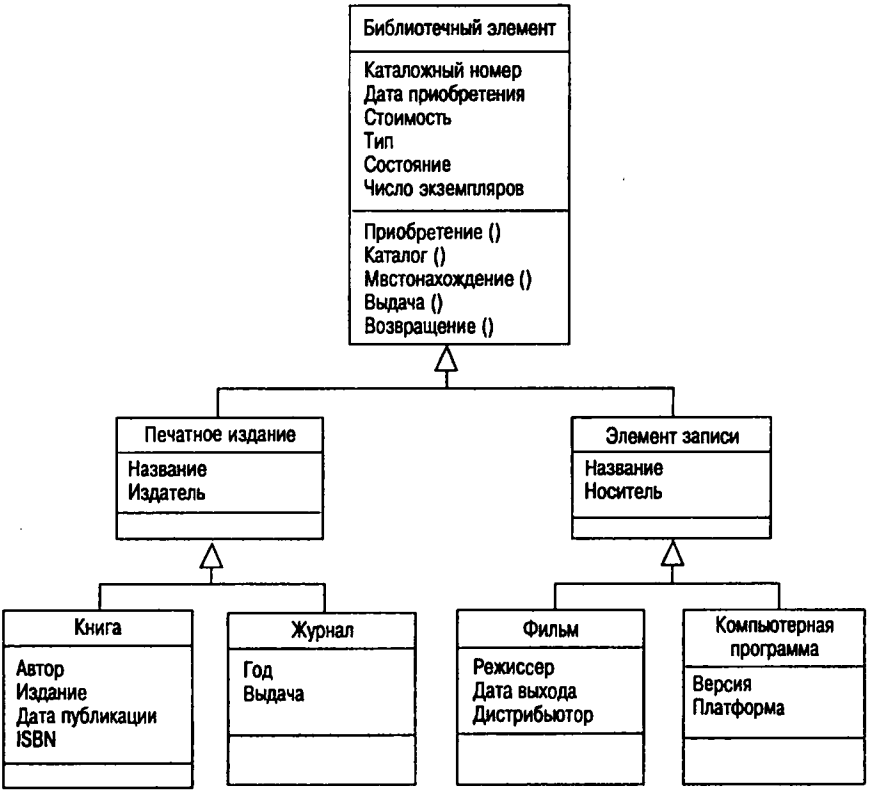


Рис. 7.8. Часть иерархии классов для библиотечной системы

Различные методы объектно-ориентированного анализа были предложены в 1990-х годах Бучем (Booch, [54]), Кодом и Джордоном (Coad and Yourdon, [74]), а также Рамбо

(Rumbaugh, [302]). Эти методы имели много общего, поэтому три главных разработчика – Буч, Рамбо и Якобсон (Jacobson) – решили интегрировать их для разработки унифицированного метода [304]. В результате разработанный ими унифицированный язык моделирования (Unified Modeling Language – UML) стал фактическим стандартом для моделирования объектов. UML предлагает нотации для различных типов системных моделей. Вы уже видели модели вариантов использования и диаграммы последовательностей в главе 5, а также диаграммы состояний ранее в этой главе.

В UML класс объектов представлен вертикально ориентированным прямоугольником с тремя секциями (рис. 7.8).

1. В верхней секции располагается имя класса.
2. Атрибуты класса находятся в средней секции.
3. Операции, связанные с классом, приводятся в нижней секции прямоугольника.

Поскольку полностью описать UML в данной книге нет возможности, здесь на объектных моделях будет показано, как классифицируются объекты, как наследуются атрибуты и операции от других объектов, а также будут приведены модели агрегирования и простые поведенческие модели.

7.4.1. Модели наследования

Важным этапом объектно-ориентированного моделирования является определение классов объектов, которые затем систематизируются. Это подразумевает создание схемы классификации, которая показывает, как классы объектов связаны друг с другом посредством общих атрибутов и сервисов.

Схема классификации организована в виде иерархии наследования, на вершине которой представлены наиболее общие классы объектов. Более специализированные объекты наследуют их атрибуты и сервисы. Эти объекты могут иметь собственные атрибуты и сервисы.

На рис. 7.8 показана часть упрощенной иерархии классов для модели библиотечной системы. Эта иерархия дает информацию о библиотечных элементах. Библиотека содержит различные типы элементов: книги, музыкальные записи, фильмы, журналы, газеты и т.д. На рис. 7.8 наиболее общий элемент расположен на вершине иерархического дерева и имеет атрибуты и сервисы, общие для всех библиотечных элементов. Они наследуются классами Печатное издание и Элемент записи, имеющими собственные атрибуты, которые затем наследуются элементами более низкого уровня.

На рис. 7.9 приведен пример другой иерархии наследования, которая также может быть частью библиотечной модели. Здесь показаны пользователи библиотеки, разбитые два класса: читатели, имеющие право выносить книги, и читатели, которые могут читать книги только в библиотеке без права выноса.

В нотации UML наследования показываются сверху вниз, как принято в других объектно-ориентированных нотациях. Здесь стрелка (с окончанием в виде треугольника) выходит из класса, который наследует атрибуты и операции, и направлена к родительскому классу. Отметим, что в UML вместо термина "наследование" чаще используется термин "обобщение".

На рис. 7.8 и 7.9 показаны иерархии классов объектов, где каждый класс наследует атрибуты и операции от одного родительского класса. В моделях множественного наследования классы могут имеет нескольких родителей. Тогда наследуются атрибуты и сервисы от каждого родительского класса. На рис. 7.10 показан пример модели множественного наследования, которая также является частью библиотечной модели.

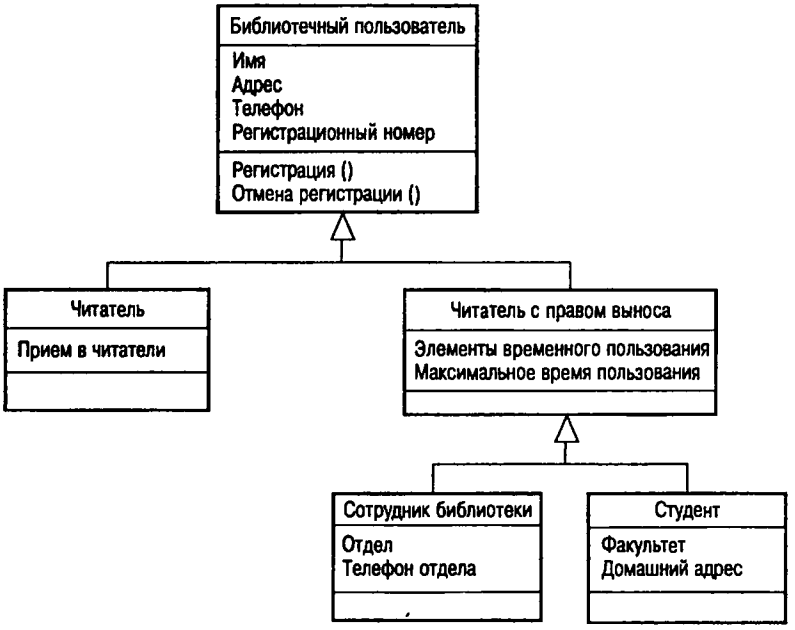


Рис. 7.9. Иерархия классов пользователей

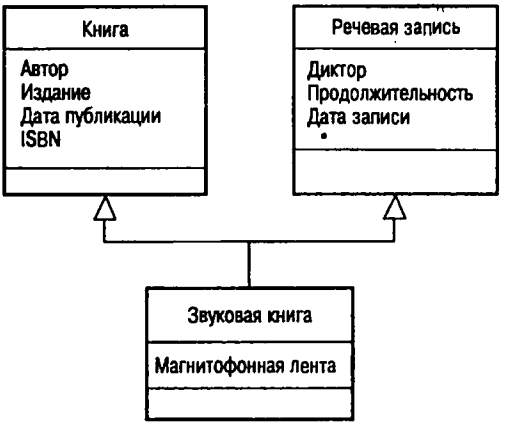


Рис. 7.10. Множественное наследование

Основной проблемой множественного наследования является разработка такой схемы, где объекты не наследуют ненужные атрибуты. Другая проблема – сложность реорганизации схемы наследования при разрешении конфликта, когда несколько родительских классов имеют атрибуты с одним и тем же именем, но разными значениями. На уровне системного моделирования такие проблемы решаются относительно просто – путем изменения объектной модели вручную. Эти проблемы служат причиной многих ошибок в объектно-ориентированном программировании.

7.4.2. Агрегирование объектов

Подобно тому как одни объекты получают атрибуты и сервисы посредством связей от других объектов, другие объекты могут создаваться из нескольких объектов. Иными словами, объект агрегируется из совокупности других объектов. Классы, представляющие такие объекты, можно смоделировать, используя модель агрегирования объектов, подобную показанной на рис. 7.11. В этом примере смоделирован библиотечный элемент, который является пакетом учебных материалов университетского курса. Этот пакет содержит записи лекций, упражнения, примеры решений, копии диапозитивов, используемых на лекциях, видеозаписи.

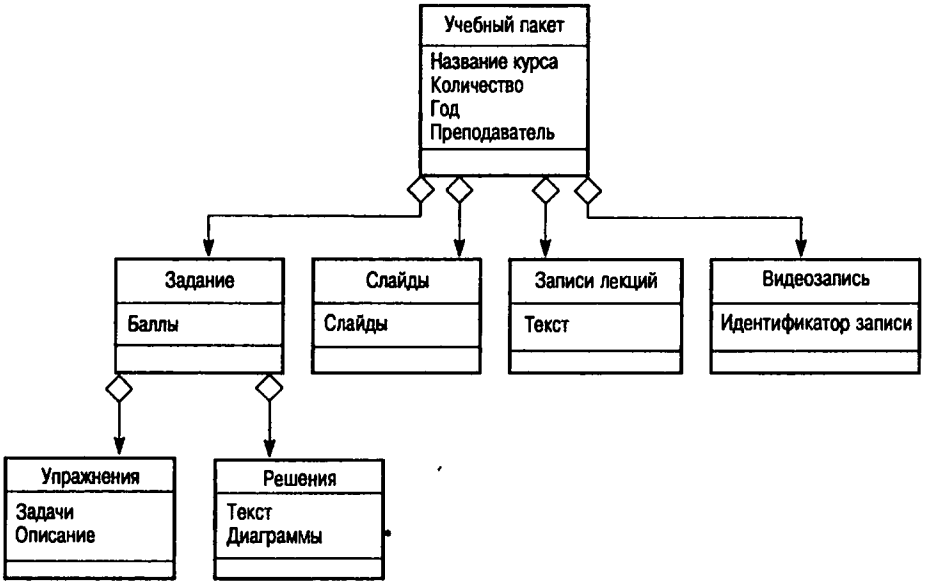


Рис. 7.11. Модель агрегирования объектов

В UML для показа агрегирования объектов используется связь с окончанием ромбовидной формы. Вот как можно прочитать рис. 7.11: "Учебный пакет, составленный из нескольких заданий, пакета слайдов, записей лекций и видеозаписей".

7.4.3. Моделирование поведения объектов

Модели поведения объектов показывают операции, выполняемые объектами. В UML поведение объектов моделируется посредством сценариев, которые основаны на вариантах использования, рассмотренных в главе 6, где приведен пример моделирования поведения, показывающий последовательность управления библиотечным каталогом (см. рис. 6.11). Кроме диаграмм последовательностей в UML предусмотрены также кооперативные диаграммы (collaboration diagrams[1]), показывающие последовательность сообщений, которыми обмениваются объекты. Они подобны диаграммам последовательностей, и здесь я на них не останавливаюсь.

[1] *Эти диаграммы в русской литературе также называют диаграммами сотрудничества.* – Прим. ред.

Я покажу диаграммы последовательностей в качестве модели поведения пользователя при получении библиотечного элемента в электронном виде. Например, пакет учебных материалов (см. рис. 7.11) может храниться в электронном виде и загружаться на компьютер студента.

На рис. 7.12 показана диаграмма последовательностей, в верхней части которой расположены объекты. Операции обозначаются помеченными стрелками, а последовательность операций читается сверху вниз. В этом сценарии библиотечный пользователь сначала обращается к электронному каталогу, чтобы увидеть, доступен ли нужный элемент, и, если доступен, запрашивает его. Для соблюдения авторских прав пользователь должен подтвердить свое согласие с лицензией на запрашиваемый материал, если, конечно, это предусмотрено правилами пользования данным материалом. Затем выбранный элемент пересылается на сетевой сервер для сжатия, после чего отсылается библиотечному пользователю.

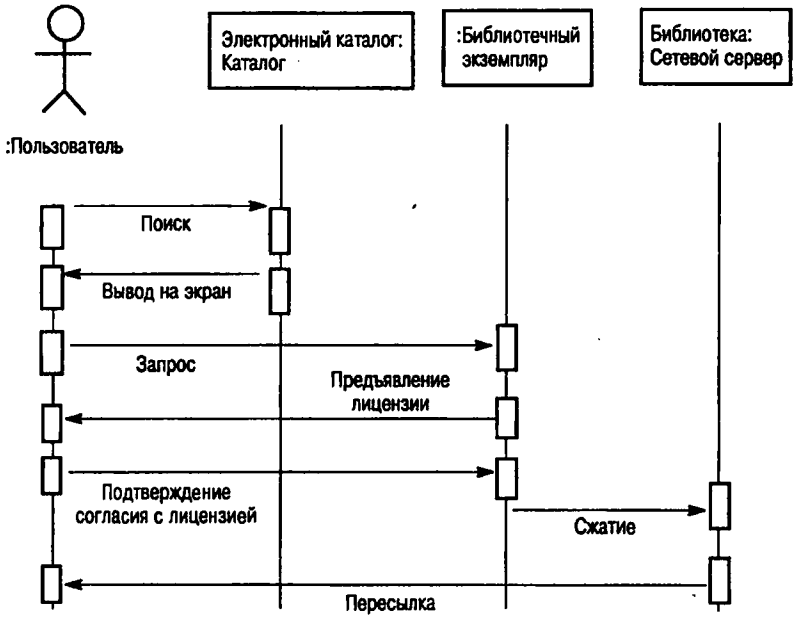


Рис. 7.12. Выдача электронных библиотечных элементов

7.5. Инструментальные CASE-средства

Это пакет программных средств, который поддерживает отдельные этапы процесса разработки программного обеспечения: проектирование, написание программного кода или тестирование. Преимущество группирования CASE-средств в инструментальный пакет заключается в том, что, работая вместе, они обеспечивают более всестороннюю поддержку процесса разработки ПО, чем могут предложить отдельные инструментальные средства. Общие сервисы могут вызываться всеми средствами. Инструментальные средства можно объединить в пакет с помощью общих файлов, репозитория или общей структуры данных.

Инструментальные средства анализа и проектирования ПО созданы для поддержки моделирования систем на этапах анализа и проектирования процесса разработки программного обеспечения. Они поддерживают создание, редактирование и анализ графических нотаций, используемых в структурных методах. Инструментальные средства анализа и проектирования часто поддерживают только определенные методы проектирования и анализа, например объектно-ориентированные. Другие инструментальные средства являются общими системами редактирования диаграмм многих типов, которые используются разными методами проектирования и анализа. Инструментальные средства, ориентированные на определенные методы, обычно автоматически поддерживают правила и базовые принципы этих методов, что позволяет выполнять автоматический контроль диаграмм.

На рис. 7.13 показана схема пакета инструментальных средств поддержки анализа и проектирования ПО. Инструментальные средства обычно объединяются через общий репозиторий, структура которого является собственностью разработчика пакета инструментальных средств. Пакеты инструментальных средств обычно закрыты, т.е. не рассчитаны на добавление пользователями собственных инструментов или на изменение средств пакета.

Рис. 7.13. Пакет инструментальных средств для анализа и проектирования ПО

Ниже перечислены средства, которые входят в пакет инструментальных средств, показанный на рис. 7.13.

1. *Редакторы диаграмм* предназначены для создания диаграмм потоков данных, иерархий объектов, диаграмм "сущность–связь" и т.д. Эти редакторы не только имеют средства рисования, но и поддерживают различные типы объектов, используемые в диаграммах.
2. *Средства проектирования, анализа и проверки* выполняют проектирование ПО и создают отчет об ошибках и дефектах в системной архитектуре. Они могут работать совместно с системой редактирования, поэтому обнаруженные ошибки можно устранить на ранней стадии процесса проектирования.
3. *Центральный репозиторий* позволяет проектировщику найти нужный проект и соответствующую проектную информацию.

4. *Словарь данных* хранит информацию об объектах, которые используются в структуре системы.
5. *Средства генерирования отчетов* на основе информации из центрального репозитория автоматически генерируют системную документацию.
6. *Средства создания форм* определяют форматы документов и экранных форм.
7. *Средства импортирования и экспортирования* позволяют обмениваться информацией из центрального репозитория различным инструментальным средствам.
8. *Генераторы программного кода* автоматически генерируют программы на основе проектов, хранящихся в центральном репозитории.

В некоторых случаях возможно генерировать программы или фрагменты программ на основе информации, представленной в системной модели. Генераторы кода, которые включены в пакеты инструментальных средств, могут генерировать код на таких языках, как Java, C++ или C. Поскольку в моделях не предусмотрена детализация низкого уровня, генератор программного кода не в состоянии сгенерировать законченную систему. Обычно необходимы программисты для завершения автоматически сгенерированных программ.

Некоторые пакеты инструментальных средств анализа и проектирования предназначены для поддержки методов разработки программных приложений деловой сферы. Обычно для создания общего репозитория инструментов они используют системы баз данных типа Sybase или Oracle. Эти пакеты инструментальных средств содержат большое количество средств языков программирования четвертого поколения, предназначенных для генерирования программного кода на основе системной архитектуры, они также могут генерировать базы данных с использованием языков программирования четвертого поколения.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Модель — это абстрактное представление системы, в котором игнорируются некоторые детали системы. Могут быть разработаны дополнительные модели системы, в которых представлена различная информация относительно системы.
- Модели рабочего окружения системы показывают, как разрабатываемая система взаимодействует с другими системами окружения. Для этого могут использоваться архитектурные модели, модели процессов и потоков данных.
- Диаграммы потоков данных используются для моделирования процесса обработки данных, выполняемого системой.
- Модель конечного автомата (диаграмма состояний) моделирует поведение системы в зависимости от внутренних или внешних событий.
- Семантические модели данных описывают логические структуры данных, импортируемых и экспортируемых системой. Эти модели отображают системные сущности, их атрибуты и связи между ними. Они могут дополняться словарями данных, где приведено более детальное описание данных.
- Объектные модели представляют системные сущности, их классификацию и агрегирование. Объектные модели включают модели наследования, агрегирования и поведенческие модели.
- Пакеты инструментальных CASE-средств поддерживают разработку системных моделей, обеспечивая их редактирование и проверку, а также средства создания отчетов и документирования.

Упражнения

7.1. Разработайте модель рабочего окружения для информационной системы больницы. Модель должна предусматривать ввод данных о новых пациентах и систему хранения рентгеновских снимков.

7.2. Создайте модель обработки данных в системе электронной почты. Необходимо отдельно смоделировать отправку почты и ее получение.

7.3. Нарисуйте модель конечного автомата управляющей системы:

- для автоматической стиральной машины, которая имеет различные программы для разных типов белья;
- для программного обеспечения проигрывателя компакт-дисков;
- для телефонного автоответчика, который регистрирует входные сообщения и показывает число принятых сообщений на дисплее. Система должна соединять владельца телефона с абонентом после ввода им последовательности чисел (телефонного номера абонента), а также, имея записанные сообщения, повторять их по телефону.

7.4. Разработайте модель классов объектов для системы электронной почты. Если вы выполнили упражнение 7.3, опишите различия и сходства между моделью обработки данных и объектной моделью.

7.5. Используя подход "сущность-связь", опишите возможную модель данных для системы библиотечного каталога, представленную в этой главе (см. рис. 7.8).

7.6. Разработайте объектную модель, включающую диаграммы иерархии классов и агрегирования, и показывающую основные элементы системы персонального компьютера и его программного обеспечения.

7.7. Разработайте диаграмму последовательностей, которая показывает действия студента, регистрирующегося на определенный курс в университете. Курс может иметь ограниченное число мест, поэтому процесс регистрации должен проверять количество доступных мест. Предположите, что студент обращается к электронному каталогу курсов, чтобы выяснить количество доступных мест.

7.8. Опишите три действия, выполняемых при моделировании систем, которые могут быть поддержаны пакетом инструментальных CASE-средств при выполнении некоторых методов анализа систем. Опишите три действия, которые невозможно легко автоматизировать.

8

Прототипирование программных систем

Цели

Цель настоящей главы — показать, как прототипирова ние используется в процессе разработки программного обеспечения, и описать различные подходы к разработ ке прототипов. Прочитав эту главу, вы должны:

- понять роль прототипирования в процессе раз работки ПО;
- знать различие между эволюционным и экспери ментальным прототипированием;
- освоить три метода разработки прототипов: с использованием языков программирования вы сокого уровня, на основе баз данных и с повтор ным использованием программных компонентов
- понять, почему прототипирование — наиболее эффективная и удобная технология проектирова ния и разработки пользовательских интерфейсов.

Содержание

Заказчикам программного обеспечения и конечным пользователям обычно сложно четко сформулировать требования к разрабатываемой программной системе. Трудно предвидеть, как система будет влиять на трудовой процесс, как она будет взаимодействовать с другими системами и какие операции, выполняемые пользователями, необходимо автоматизировать. Тщательный анализ требований помогает уменьшить неопределеиность относительно того, что система должна делать. Однако реально проверить требования, прежде чем их утвердить, практически невозможно. В этой ситуации может помочь прототип системы.

Прототип является начальной версией программной системы, которая используется для демонстрации концепций, заложенных в системе, проверки вариантов требований, а также поиска проблем, которые могут возникнуть как в ходе разработки, так и при эксплуатации системы, и возможных вариантов их решения. Очень важна быстрая разработка прототипа системы, чтобы пользователи могли начать экспериментировать с ним как можно раньше.

Прототип ПО помогает на двух этапах процесса разработки системных требований.

1. *Постановка требований.* Пользователи могут экспериментировать с системными прототипами, что позволяет им проверять, как будет работать система. Пользователи получают новые идеи для постановки требований, могут определить сильные и слабые стороны ПО. В результате могут сформироваться новые требования.
2. *Проверка требований.* Прототип позволяет обнаружить ошибки и упущения в ранее принятых требованиях. Например, системные функции, определенные в требованиях, могут быть полезными и нужными (с точки зрения пользователя). Однако в процессе применения этих функций совместно с другими функциями пользователи могут изменить первоначальное мнение о них. В результате требования к системе изменятся, отражая измененное понимание пользователями системных функций.

Прототипирование можно использовать при анализе рисков и на начальном этапе разработки планов управления программным проектом (см. главу 4). Основной опасностью при разработке ПО являются ошибки и упущения в требованиях. Затраты на устранение ошибок в требованиях на более поздних стадиях процесса разработки могут быть очень высокими. Эксперименты показывают [48], что прототипирование уменьшает число проблем, связанных с разработкой требований. Кроме того, прототипирование уменьшает общую стоимость разработки системы. По этим причинам оно часто используется в процессе разработки требований.

Однако различие между прототипированием, как отдельным этапом процесса разработки ПО, и разработкой основной программной системы неочевидно. В настоящее время многие системы разрабатываются с использованием эволюционного подхода, когда быстро создается первоначальная версия системы, которая затем постепенно изменяется до ее окончательного варианта. При этом часто используются методы быстрой разработки приложений, которые также можно использовать при создании прототипов. Эти вопросы обсуждаются в разделе 8.1.

Наряду с тем что прототипы помогают формировать требования, они имеют и другие достоинства.

1. Различное толкование требований разработчиками ПО и пользователями можно выявить при демонстрации действующего прототипа системы.
2. В процессе создания прототипа разработчики могут выявить неполные или несогласованные требования.
3. Работая, хотя и ограниченно, в виде прототипа, система может продемонстрировать свои слабые и сильные стороны.

4. Прототип может служить основой для написания спецификации высококачественной системы. Разработка прототипа обычно ведет к улучшению спецификации системы.

Действующий прототип может также использоваться для других целей [178].

1. *Обучение пользователя.* Прототип системы можно использовать для обучения персонала перед поставкой окончательного варианта системы.
2. *Тестирование системы.* Прототипы позволяют "прокручивать" тесты. Один и тот же тест запускается на прототипе и на системе. Если получаются одинаковые результаты, это означает, что тест не обнаружил дефектов в системе. Если результаты отличаются, то необходимо исследовать причины различия, что позволяет выявить возможные ошибки в системе.

На основе изучения 39 различных программных проектов, использовавших прототипирование, в работе [133] сделан вывод, что эффективность применения прототипов при разработке ПО состоит в следующем.

1. Улучшаются эксплуатационные качества системы.
2. Система больше соответствует потребностям пользователей.
3. Системная архитектура становится более совершенной.
4. Сопровождение системы упрощается и становится более удобным.
5. Сокращаются расходы на разработку системы.

Эти исследования показывают, что улучшение эксплуатационных качеств системы и увеличение соответствия системы потребностям пользователя не требуют увеличения общей стоимости разработки системы. Прототипирование обычно повышает стоимость начальных этапов разработки ПО, но снижает затраты на более поздних этапах.

Модель процесса разработки прототипа показана на рис. 8.1. На первом этапе данного процесса определяются назначение прототипа и цель прототипирования. Целью может быть разработка макета пользовательского интерфейса, проверка функциональных системных требований или демонстрация реализуемости системы для руководства. Один и тот же прототип не может служить одновременно всем целям. Если цели определены неточно, функции прототипа могут быть восприняты неверно.

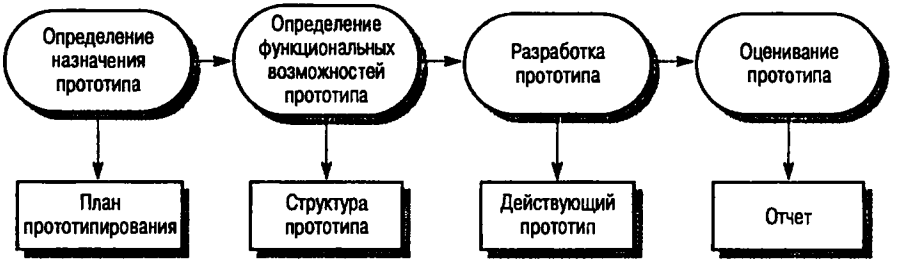


Рис. 8.1. Процесс разработки прототипа

На следующем этапе процесса разработки прототипа определяются его функциональные возможности, т.е. принимается решение о том, какие свойства системы должен отражать прототип, а какие (что, возможно, более важно) — нет. Для уменьшения затрат на создание прототипа можно исключить некоторые системные функции. Например, можно ослабить временные характеристики и требования к использованию памяти. Средства

управления и обработки ошибок могут игнорироваться либо быть элементарными, если, конечно, целью прототипирования не является модель интерфейса пользователя. Также могут быть снижены требования к надежности и качеству программ.

Заключительный этап процесса прототипирования — оценивание созданного прототипа. В работе [178] утверждается, что это наиболее важный этап процесса прототипирования. Здесь проверяется, насколько созданный прототип соответствует своему назначению и целям, а также на его основе создается план мероприятий по совершенствованию разрабатываемой системы.

8.1. Прототипирование в процессе разработки ПО

Как уже отмечалось, конечным пользователям трудно представить, как они будут использовать новую систему ПО в повседневной работе. Если система большая и сложная, то это невозможно сделать, прежде чем система будет создана и введена в эксплуатацию.

Один из способов преодоления этой трудности состоит в использовании эволюционного метода разработки систем. Это означает, что пользователю предоставляется незавершенная система, которая затем изменяется и дополняется до тех пор, пока не станут ясны все требования пользователя. В качестве альтернативы можно построить "экспериментальный" прототип, который поможет проанализировать и проверить требования. После этого создается система. На рис. 8.2 показаны оба подхода к использованию прототипов.

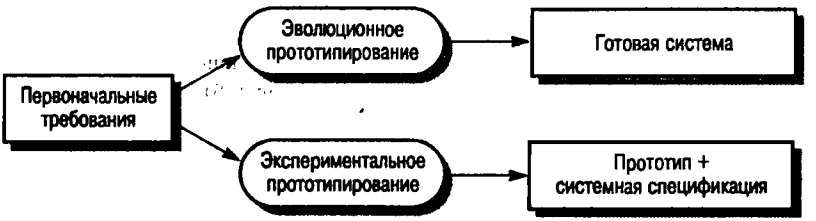


Рис. 8.2. Эволюционное и экспериментальное прототипирование

Эволюционное прототипирование начинается с построения относительно простой системы, которая реализует наиболее важные требования пользователя. По мере выявления новых требований прототип изменяется и дополняется. В конечном счете он становится той системой, которая требуется. В этом процессе не используется детальная системная спецификация, во многих случаях нет даже формального документа с системными требованиями. В настоящее время эволюционное прототипирование является обычной технологией разработки программных систем, которая широко используется при разработке Web-узлов и приложений электронной коммерции.

В противоположность эволюционному подходу метод экспериментального прототипирования предназначен для разработки и уточнения системной спецификации. Прототип создается, оценивается и модифицируется. Данные оценивания прототипа используются для дальнейшей детализации спецификации. Когда системные требования сформированы, прототип больше не нужен.

Существует различие между целями эволюционного и экспериментального прототипирования.

- Целью эволюционного прототипирования является поставка работающей системы конечному пользователю. Это означает, что необходимо начать создание системы,

реализующей требования пользователя, которые наиболее понятны и которые имеют наивысший приоритет. Требования с более низким приоритетом и нечеткие требования реализуются по запросам пользователей.

- Целью экспериментального прототипирования является проверка и формирование системных требований. Здесь сначала создается прототип, реализующий те требования, которые сформулированы нечетко и с которыми необходимо "разобраться". Требования, которые сформулированы четко и понятно, не нуждаются в прототипировании.

Другое важное различие между этими подходами касается управления качеством разрабатываемой системы. Экспериментальные прототипы имеют очень короткий срок жизни. Они быстро меняются и для них высокая эксплуатационная надежность не требуется. Для экспериментального прототипа допускается пониженная эффективность и безотказность, поскольку прототип должен выполнить только свою основную функцию – помочь в понимании требований.

В противоположность этому прототипы, которые эволюционируют в законченную систему, должны быть разработаны с такими же стандартами качества, что и любое другое программное обеспечение. Они должны иметь устойчивую структуру и высокую эксплуатационную надежность. Они должны быть безотказны, эффективны и отвечать соответствующим стандартам.

8.1.1. Эволюционное прототипирование

В основе эволюционного прототипирования лежит идея разработки первоначальной версии системы, демонстрации ее пользователям и последующей модификации вплоть до получения системы, отвечающей всем требованиям (рис. 8.3). Такой подход сначала использовался для разработки систем, которые трудно или невозможно специфицировать (например, систем искусственного интеллекта). В настоящее время он становится основной методикой при разработке программных систем. Эволюционное прототипирование имеет много общего с методами быстрой разработки приложений и часто входит в эти методы как их составная часть [238, 346, 327, 6*].

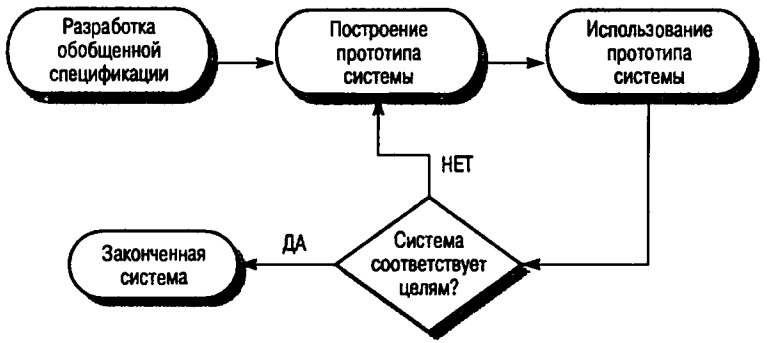


Рис. 8.3. Эволюционное прототипирование

Этот метод прототипирования имеет два основных преимущества.

1. *Ускорение разработки системы.* Как указывалось во введении, современные темпы изменений в деловой сфере требуют быстрых изменений программного обеспечения. В некоторых случаях быстрая поставка ПО, удобство и простота его использования

более важны, чем полный спектр функциональных возможностей системы или долгосрочные возможности ее сопровождения.

2. *Взаимодействие пользователя с системой.* Участие пользователей в процессе разработки означает, что в системе более полно будут учтены пользовательские требования.

Между отдельными методами быстрой разработки ПО существуют различия, но все они имеют некоторые общие свойства.

1. Этапы разработки технических требований, проектирования и реализации перемежаются. Не существует детальной системной спецификации, проектная документация обычно зависит от инструментальных средств, используемых для реализации системы. Пользовательские требования определяют только наиболее важные характеристики системы.
2. Система разрабатывается пошагово. Конечные пользователи и другие лица, формирующие требования, участвуют на каждом шаге проектирования и оценивания новой версии системы. Они могут предлагать изменения и новые требования, которые будут реализованы в следующей версии системы.
3. Применение методов быстрой разработки систем (см. раздел 8.2). Они могут использовать инструментальные CASE-средства и языки четвертого поколения.
4. Пользовательский интерфейс системы обычно создается с использованием интерактивных систем разработки (см. раздел 8.3), которые позволяют быстро спроектировать и создать интерфейс.

Эволюционное прототипирование и методы, основанные на использовании детальной системной спецификации, отличаются подходами к верификации и аттестации систем. Верификация – процесс проверки системы на соответствие спецификации. Поскольку для прототипа не создается подробной спецификации, его верификация невозможна.

Аттестация системы должна показать, что программа соответствует тем целям, для которых она создавалась. Аттестацию также трудно провести без детальной спецификации, поскольку нет четких формулировок целей. Конечные пользователи, участвующие в процессе разработки, могут быть удовлетворены системой, в то время как другие пользователи – неудовлетворены, поскольку система не полностью соответствует тем целям, которые они неявно перед ней поставили.

Верификацию и аттестацию системы, разработанной с использованием эволюционного прототипирования, можно осуществить, если она в достаточной степени соответствует поставленной цели и своему назначению. Это соответствие, конечно, нельзя измерить, можно сделать лишь субъективные оценки. Такой подход, как будет показано ниже, может породить проблемы, если программная система создается сторонними организациями-разработчиками.

Существует три основные проблемы эволюционного прототипирования, которые необходимо учитывать, особенно при разработке больших систем с длительным сроком жизненного цикла.

1. *Проблемы управления.* Структура управления разработкой программных систем строится в соответствии с утвержденной моделью процесса создания ПО, где для оценивания очередного этапа разработки используются специальные контрольные проектные элементы (см. главу 4). Прототипы эволюционируют настолько быстро, что создавать контрольные элементы становится нерентабельно. Кроме того, быстрая разработка прототипа может потребовать применения новых технологий. В этом случае может возникнуть необходимость привлечения специалистов с более высокой квалификацией.

2. *Проблемы сопровождения системы.* Из-за непрерывных изменений в прототипах изменяется также структура системы. Это означает, что система будет трудна для понимания всем, кроме первоначальных разработчиков. Кроме того, может устареть специальная технология быстрой разработки, которая использовалась при создании прототипов. Поэтому могут возникнуть трудности при поиске людей, которые имеют знания, необходимые для сопровождения системы.
3. *Проблемы заключения контрактов.* Обычно контракт на разработку систем между заказчиком и разработчиками ПО основывается на системной спецификации. При отсутствии таковой трудно составить контракт на разработку системы. Для заказчика может быть невыгоден контракт, по которому приходится просто платить разработчикам за время, потраченное на разработку проекта; также маловероятно, что разработчики согласятся на контракт с фиксированной ценой, поскольку они не могут предвидеть все прототипы, которые потребуется создать в процессе разработки системы.

Из этих проблем вытекает, что заказчики должны понимать, насколько эффективно эволюционное прототипирование в качестве метода разработки ПО. Этот метод позволяет быстро создавать системы малого и среднего размера, при этом стоимость разработки снижается, а качество повышается. Если к процессу разработки привлекаются конечные пользователи, то, вероятно, система будет соответствовать их реальным потребностям. Однако организации-разработчики, использующие этот метод, должны учитывать, что жизненный цикл таких систем будет относительно короток. При возрастании проблем с сопровождением систему необходимо заменить или полностью переписать. Для больших систем, когда к разработке привлекаются субподрядчики, на первый план выходят проблемы управления эволюционным прототипированием. В этом случае лучше применять экспериментальное прототипирование.

Пошаговая разработка (рис. 8.4) позволяет избежать некоторых проблем, характерных для эволюционного прототипирования. Общая архитектура системы, определенная на раннем этапе ее разработки, выступает в роли системного каркаса. Компоненты системы разрабатываются пошагово, затем включаются в этот каркас. Если компоненты аттестованы и включены в каркас, ни архитектура, ни компоненты уже не меняются, за исключением случая, когда обнаруживаются ошибки.

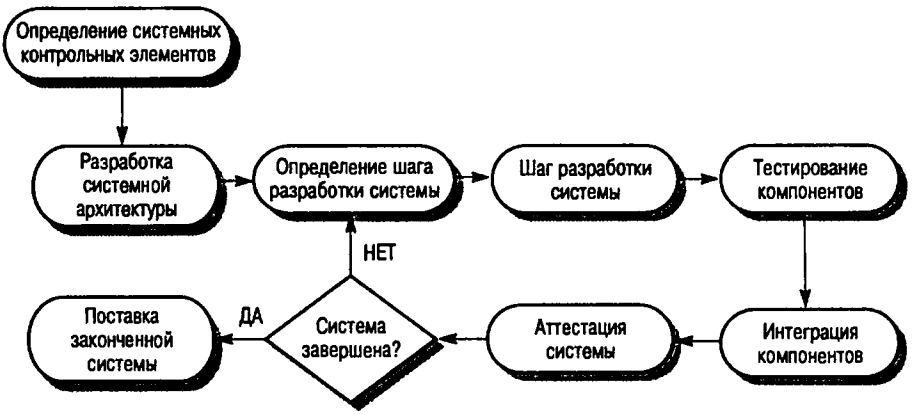


Рис. 8.4. Пошаговый процесс разработки

Процесс пошаговой разработки более управляем, чем эволюционное прототипирование, поскольку следует обычным стандартам разработки ПО. Здесь планы и документация создаются для каждого шага разработки системы, что уменьшает количество ошибок. Как только системные компоненты интегрированы в каркас, их интерфейсы больше не изменяются.

8.1.2. Экспериментальное прототипирование

Модель процесса разработки ПО, основанная на экспериментальном прототипировании, показана на рис. 8.5. В этой модели расширен этап анализа требований в целях уменьшения общих затрат на разработку. Основное назначение прототипа — сделать понятными требования и предоставить дополнительную информацию для оценки рисков. После этого прототип больше не используется и не участвует в дальнейшем процессе разработки системы.

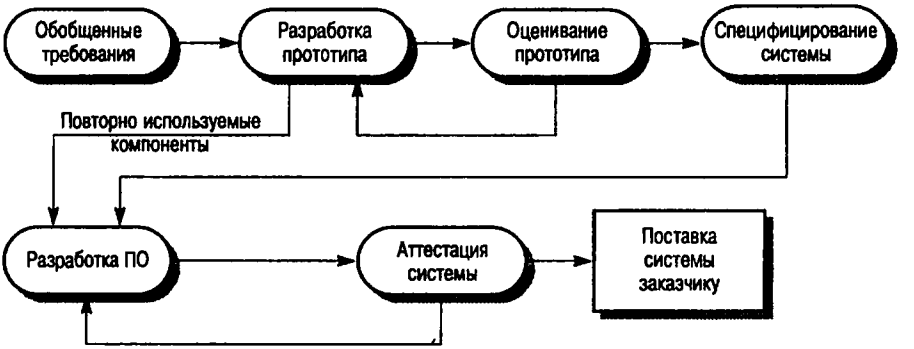


Рис. 8.5. Разработка ПО с использованием экспериментальных прототипов

Этот метод прототипирования обычно используется для разработки аппаратных систем. Прежде чем будет начата дорогостоящая разработка системы, создается макет (прототип), который используется для проверки структуры системы. Электронный макет системы создается с использованием готовых компонентов, что позволяет разработать версию системы до того, как будут вложены денежные средства в разработку специализированных интегральных схем.

Экспериментальный прототип программных систем обычно не используется для проверки архитектуры системы, он помогает разработать системные требования. Прототип часто совершенно не похож на конечную систему. Система разрабатывается по возможности быстро, поэтому для ускорения формирования требований используется упрощенный прототип системы. В экспериментальный прототип закладываются только обязательные системные функции, стандарты качества для прототипа могут быть снижены, критерии эффективности игнорируются. Язык программирования прототипа часто отличается от языка программирования, на котором будет создаваться окончательный вариант системы.

В модели процесса разработки ПО, показанной на рис. 8.5, предполагается, что прототип разрабатывается исходя из обобщенных системных требований, далее над прототипом проводятся эксперименты и он изменяется до тех пор, пока его функциональные возможности не удовлетворят заказчика. После этого на основе прототипа детализируются системные требования, реализуется обычная для организации-разработчика технология разработки ПО и система доводится до окончательной версии. Некоторые компоненты прототипа могут использоваться в системе, поэтому стоимость разработки может быть снижена.

В описываемой модели разработки ПО основная проблема состоит в том, что экспериментальный прототип может не соответствовать конечной системе, поставляемой заказчику. Специалист, тестирующий прототип, может иметь собственные интересы к системе, не типичные для ее пользователей. Время тестирования прототипа может быть недостаточным для его полного оценивания. Если прототип работает медленно, эксперты могут внести в него изменения, которые ускоряют работу, но не будут совпадать со средствами ускорения работы конечной системы.

Разработчики иногда подвергаются давлению менеджеров для ускорения работы над прототипом, особенно если намечается задержка в поставке окончательной версии системы. Обычно такое "ускорение" порождает ряд проблем.

1. Невозможно быстро настроить прототип для выполнения таких нефункциональных требований, как производительность, защищенность, устойчивость к сбоям и безотказность, которые игнорировались во время разработки прототипа.
2. Частые изменение во время разработки неизбежно приводят к тому, что прототип плохо документирован. Для разработки прототипа используется только спецификация системной архитектуры. Этого недостаточно для долговременного сопровождения системы.
3. Изменения, сделанные во время разработки прототипа могут нарушить архитектуру системы. Ее обслуживание будет сложным и дорогостоящим.
4. В процессе разработки прототипа ослабляются стандарты качества.

Чтобы быть полезными в процессе разработки требований, экспериментальные прототипы не обязательно должны выполнять роль реальных макетов систем. Бумажные формы, имитирующие пользовательские интерфейсы систем [292], показали свою эффективность при формировании требований пользователя, в уточнении проекта интерфейса и при создании сценариев работы конечного пользователя. Они очень дешевы в разработке и могут быть созданы за несколько дней. Расширением этой методики является макет пользовательского интерфейса "Wizard of Oz" (Волшебник страны Оз). Пользователи взаимодействуют с этим интерфейсом, но их запросы направлены к специалисту, который интерпретирует их и имитирует соответствующую реакцию. Подобные подходы к прототипированию рассмотрены в работе [321].

8.2. Технологии быстрого прототипирования

Эти технологии рассчитаны главным образом на обеспечение быстрой разработки прототипов, а не на такие их системные характеристики, как производительность, удобство эксплуатации или безотказность. Существует три основных метода быстрой разработки прототипов.

1. Разработка с применением динамических языков высокого уровня.
2. Использование языков программирования баз данных.
3. Сборка приложений с повторным использованием компонентов.

Для удобства эти методы описаны в отдельных разделах. Но на практике они часто совместно используются при разработке прототипов систем. Например, язык программирования баз данных может применяться для извлечения данных с их последующей обработкой с помощью повторно используемых компонентов. Интерфейс пользователя системы можно разработать, используя визуальное программирование. В статье [224] описано смешанное применение этих методов при создании прототипа управляющей системы.

В настоящее время разработка прототипов обычно опирается на набор инструментов, поддерживающих по крайней мере два из этих методов. Например, система Smalltalk VisualWorks поддерживает язык очень высокого уровня и обеспечивает повторное использование компонентов. Пакет Lotus Notes включает поддержку программирования баз данных с помощью языка высокого уровня и повторное использование компонентов, которые могут обеспечить операции над базой данных.

Большинство систем прототипирования сегодня поддерживают визуальное программирование, при котором некоторые части или весь прототип разрабатываются в интерактивном режиме. Вместо последовательного написания программ разработчик прототипа предпочитает работать с графическими пиктограммами, представляющими функции, данные или компоненты интерфейса пользователя, и соответствующими сценариями управления этими пиктограммами. Программа, готовая к исполнению, генерируется автоматически из визуального представления системы. Это упрощает разработку программы и уменьшает затраты на прототипирование. Более подробно визуальное программирование рассматривается в разделе 8.2.3.

8.2.1. Применение динамических языков высокого уровня

Динамические языки высокого уровня — это языки программирования, которые имеют мощные средства контроля данных во время выполнения программы. Они упрощают разработку программ, так как уменьшают число проблем, связанных с распределением памяти и управлением ею. Такие языки имеют средства, которые обычно должны быть построены из более примитивных конструкций в языках, подобных Ada или C. Примеры языков очень высокого уровня — Lisp (основанный на структурах списков), Prolog (основанный на алгебре логики) и Smalltalk (основанный на объектах).

До недавнего времени динамические языки высокого уровня широко не использовались для разработки больших систем, поскольку они нуждаются в основательных средствах динамической поддержки. Эти средства увеличивали объем необходимой памяти и уменьшали скорость выполнения программ, написанных на этих языках. Однако возрастание мощности и снижение стоимости компьютерного оборудования сделало эти факторы не столь существенными.

Таким образом, для многих деловых приложений эти языки могут заменить такие традиционные языки программирования, как C, COBOL и Ada. Язык Java, несомненно, является основным языком разработки, имеющим корни в языке C++, но с включением многих средств языка Smalltalk наподобие платформенной независимости и автоматического управления памятью. Язык Java объединяет в себе многие преимущества языков высокого уровня, совмещая это с точностью и возможностью оптимизации выполнения, обычно предлагаемой языками третьего поколения. В языке Java много компонентов, доступных для повторного использования, все это делает его подходящим для эволюционного прототипирования.

В табл. 8.1 представлены динамические языки, которые более всего используются при разработке прототипов. При выборе языка для написания прототипа необходимо ответить на ряд вопросов.

1. *Каков тип разрабатываемого приложения?* Как показано в табл. 8.1, для каждого типа приложения можно применить несколько различных языков. Если необходим прототип приложения, которое обрабатывает данных на естественном языке, то языки Lisp или Prolog более подходят, чем Java или Smalltalk.

2. *Каков тип взаимодействия с пользователем?* Различные языки обеспечивают разные типы взаимодействия с пользователем. Некоторые языки, такие как Smalltalk и Java, хорошо интегрируются с Web-броузерами, в то время как язык Prolog лучше всего подходит для разработки текстовых интерфейсов.
3. *Какую рабочую среду обеспечивает язык?* Развитая рабочая среда поддержки языка со своими инструментальными средствами и легким доступом к повторно используемым компонентам упрощает процесс разработки прототипа.

Таблица 8.1. Языки высокого уровня, используемые при прототипировании

Язык	Тип языка	Тип приложения
Smalltalk	Объектно-ориентированный	Интерактивные системы
Java	Объектно-ориентированный	Интерактивные системы
Prolog	Логический	Системы обработки символьной информации
Lisp	Основанный на списках	Системы обработки символьной информации

Динамические языки высокого уровня для создания прототипа можно использовать совместно, когда различные части прототипа программируются на разных языках. В работе [350] описывается разработка прототипа телефонной сетевой системы, где были использованы четыре различных языка: Prolog для макетирования баз данных, Awk [5] для составления счетов, CSP [163] для спецификации протоколов и PAISLey [351] для имитирования работы системы.

Не существует идеального языка для прототипирования больших систем, поскольку обычно различные части системы разнотипны. Преимущество многоязычного подхода в том, что для создания каждого компонента можно подобрать наиболее подходящий язык и таким образом ускорить разработку прототипа. Недостаток такого подхода в том, что трудно разработать коммуникационные связи для компонентов, написанных на разнородных языках.

8.2.2. Программирование баз данных

Эволюционная разработка в настоящее время является стандартной методикой для создания бизнес-приложений малого и среднего размера. Большинство бизнес-приложений включают в себя систему управления базой данных и обработку данных, находящихся в ней.

Для поддержки разработки таких приложений все коммерческие системы управления базами данных имеют внутренние средства программирования. Программирование баз данных выполняется на основе специализированных языков, которые имеют встроенную базу знаний и средства, необходимые для работы с базами данных. Рабочая среда поддержки языка обеспечивает инструментальные средства для создания пользовательских интерфейсов, числовых вычислений и отчетов. Термин *язык четвертого поколения* применяется как к самому языку программирования баз данных, так и к его рабочей среде.

Языки четвертого поколения успешно применяются на практике, поскольку большинство современных приложений в той или иной мере занимаются обработкой информации, заключенной в базах данных. Основные операции, выполняемые такими приложениями, — это модификация базы данных и создание отчетов на основе информации, извлеченной из базы данных. Обычно для ввода и вывода данных используются стандартные формы. Языки четвертого поколения имеют средства для создания интерактивных приложений, позволяющие пользователям вносить изменения в базу данных. Пользовательский интерфейс обычно состоит из набора стандартных форм или электронной таблицы.

Обычно рабочая среда языков четвертого поколения включает следующие инструментальные средства (рис. 8.6).

1. В качестве языка программирования баз данных (точнее, языка запросов к базе данных) обычно используется SQL [87].
2. Генератор интерфейсов используется для создания форм ввода и отображения данных.
3. Электронная таблица применяется для анализа данных и выполнения различных действий над числовой информацией.
4. Генератор отчетов предназначен для создания отчетов на основе информации, содержащейся в базе данных.

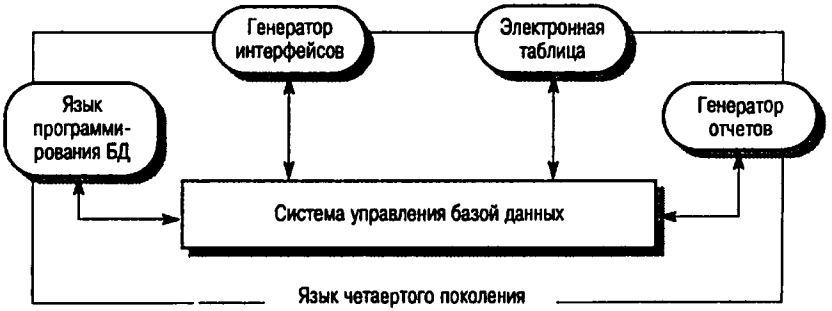


Рис. 8.6. Компоненты языка четвертого поколения

Большинство бизнес-приложений предполагают структурированные формы для ввода и вывода данных, поэтому языки четвертого поколения обеспечивают мощные средства для определения экранных форм и создания отчетов. Экранные формы часто определяются как ряд взаимосвязанных форм (в одном приложении, которое мы исследовали, было 137 различных форм), поэтому система, генерирующая экраны, должна обеспечивать следующее.

1. *Интерактивное определение форм*, когда разработчик определяет поля ввода и их организацию.
2. *Связывание форм*, когда разработчик задает определенные данные, ввод которых вызывает отображение дальнейших форм.
3. *Проверка входных данных*, когда разработчик при формировании полей форм определяет дозволенный диапазон входных величин.

В настоящее время большинством языков четвертого поколения поддерживается разработка интерфейсов баз данных, основанных на Web-броузерах. Они делают базу данных доступной с помощью Internet. Это снижает стоимость обучения и программного обеспечения и позволяет внешним пользователям иметь доступ к базе данных. Однако ограничения протоколов Internet и медленный просмотр Web-страниц делают этот метод не подходящим для систем, в которых требуется быстрое взаимодействие с пользователем.

Методы, основанные на языках четвертого поколения, могут использоваться для эволюционного прототипирования или для генерирования "одноразового" прототипа системы. Структура, которую CASE-средства накладывают на разрабатываемое приложение и сопутствующую документацию, определяет более удобное сопровождение прототипов,

чем предлагают прототипы, разработанные вручную. CASE-средства могут генерировать код SQL или код на языке низшего уровня, например COBOL. В статье [116] в кратком обзоре языков четвертого поколения описан ряд инструментальных средств этого типа.

Хотя языки четвертого поколения подходят для разработки прототипов, все же они имеют ряд недостатков, проявляющихся при разработке систем. Программы, написанные на языках четвертого поколения, как правило, выполняются медленнее подобных программ, написанных на обычных языках программирования, и требуют намного больше памяти. Например, я участвовал в эксперименте, в котором перезапись на язык C++ программы, написанной на языке четвертого поколения, привела к 50%-му сокращению необходимой памяти. Программа на C также выполнялась в 10 раз быстрее, чем аналогичная программа, написанная с использованием языка четвертого поколения.

Несмотря на то что применение языков четвертого поколения снижает стоимость разработки систем, общая сумма затрат за полный жизненный цикл таких систем пока не ясна. Их программы обычно плохо структурированы и трудны в сопровождении. Специфические проблемы могут возникать при модификации подобных систем. В настоящее время языки четвертого поколения не стандартизированы и не унифицированы, поэтому при модификации систем, скорее всего, придется переписать программы, поскольку язык, на котором они написаны, устареет.

8.2.3. Сборка приложений с повторным использованием компонентов

Время, необходимое для разработки системы, можно уменьшить, если многие части такой системы будут использованы неоднократно. Для быстрого построения прототипа необходимо иметь набор компонентов, пригодных для повторного использования, и механизм сборки системы из этих компонентов. Этот подход показан на рис. 8.7.

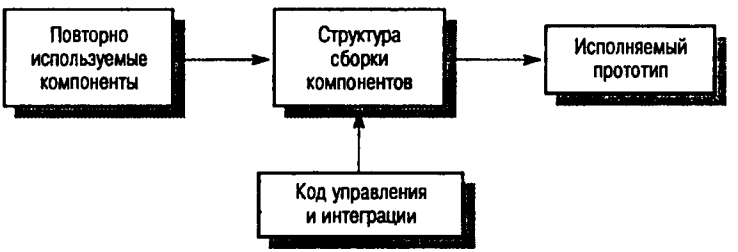


Рис. 8.7. Сборка повторно используемых компонентов

Прототипирование с повторно используемыми компонентами применяется при разработке требований, конечно, если есть подходящие компоненты. Если подходящих компонентов нет, то для реализации некоторых требований будет необходим компромиссный подход. Функциональные возможности доступных компонентов могут не точно соответствовать пользовательским требованиям. Но, с другой стороны, эти требования обычно достаточно гибкие, поэтому во многих случаях возможно создание прототипа.

Разработку прототипа с повторным использованием компонентов можно реализовать на двух уровнях.

1. Уровень приложения, когда целые прикладные системы интегрируются с прототипом так, чтобы были объединены их функциональные возможности. Например, если прототипу требуются средства обработки текста, то это можно обеспечить путем интеграции в прототип стандартной системой текстового процессора. Отметим, что приложения Microsoft Office поддерживают интеграцию со сторонними системами.

2. Уровень компонентов, когда отдельные компоненты объединяются внутри структуры, реализующей систему. Такая структура может быть создана с помощью одного из языков описания сценариев, таких, как Visual Basic, TCL/TK [267], Python [225] или Perl [337]. В качестве альтернативы могут применяться такие системы, как CORBA, DCOM или JavaBeans [311, 264, 280, 27*].

Повторно используемые приложения дают доступ ко всем своим функциональным возможностям. Если, кроме того, приложение обеспечивает создание сценариев или средства автоматизации (например, макросы Excel), они также могут использоваться для расширения функциональных возможностей прототипа.

Для понимания этого метода разработки прототипа полезен составной документ, который представляет собой схему обработки данных прототипом и который можно рассматривать как контейнер для нескольких различных объектов. Эти объекты содержат разные типы данных (такие, как таблица, диаграмма, форма), которые могут обрабатываться различными приложениями.

На рис. 8.8 представлен составной документ для прототипа системы, включающего текстовые элементы, элементы электронной таблицы и звуковые файлы. Текстовые элементы обрабатываются текстовым процессором, таблицы – электронной таблицей, а звуковые файлы – аудиопроигрывателем. Когда пользователь системы обращается к объекту определенного типа, вызывается связанное с ним приложение.

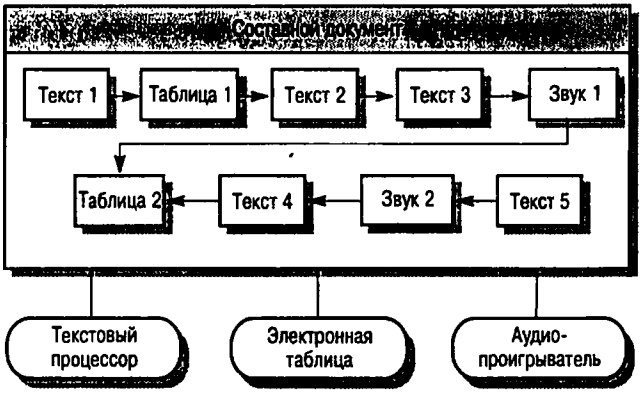


Рис. 8.8. Связывание приложений посредством составного документа

Рассмотрим прототип системы, поддерживающей управление разработкой требований (см. главу 6). Для этой системы необходимы средства фиксации требований, их хранения, создания отчетов, поиска зависимостей между требованиями и управления этими зависимостями с помощью матриц оперативного контроля. В прототипе должна быть база данных (для хранения требований), текстовый процессор (для ввода требований и создания отчетов), электронная таблица (для управления матрицами контроля) и специально написанная программа для поиска зависимостей между требованиями.

Основное преимущество описываемого подхода к прототипированию состоит в том, что многие функциональные средства прототипа можно реализовать быстро и дешево. Если пользователи, тестирующие прототип, знакомы с приложениями, интегрированными в прототип, им нет необходимости учиться использовать новые средства прототипа. Проблемы при работе с прототипом могут возникнуть только при переключении с одного

приложения на другое. Но это в значительной степени зависит от используемой операционной системы. Для организации переключения между приложениями наиболее широко используется механизм связывания и внедрения объектов OLE от Microsoft [311].

Не всегда возможно или удобно использовать целые приложения. Для создания прототипов можно использовать более "тонкие" компоненты. Это могут быть отдельные функции или объекты, которые выполняют специальные действия, например сортировку, поиск, отображение данных и т.д. Прототипирование начинается с определения общей структуры прототипа, затем компоненты интегрируются в соответствии с этой структурой. Если нет компонентов, выполняющих требующиеся функции, вместо них разрабатываются отдельные программы, которые в будущем также можно повторно использовать.

Визуальные системы разработки приложений, подобные Visual Basic, поддерживают повторное использование компонентов. Разработчики строят систему в интерактивном режиме, определяя интерфейс в виде набора экранных форм, полей, кнопок и меню. Эти элементы интерфейса именованы, и каждый из них связан со сценарием обработки событий и данных. Эти сценарии могут вызывать повторно используемые программные компоненты.

На рис. 8.9 показан экран приложения, содержащий меню (в верхней части), поля ввода (белые области слева на экране), поля вывода (затенная область слева) и кнопки, представленные скругленными прямоугольниками справа на экране. После расположения этих графических элементов на экране разработчик определяет, какие готовые компоненты будут связаны с ними, либо пишет программы, выполняющие необходимые действия. На рис. 8.9 показаны компоненты, связанные с некоторыми отображаемыми элементами экрана.

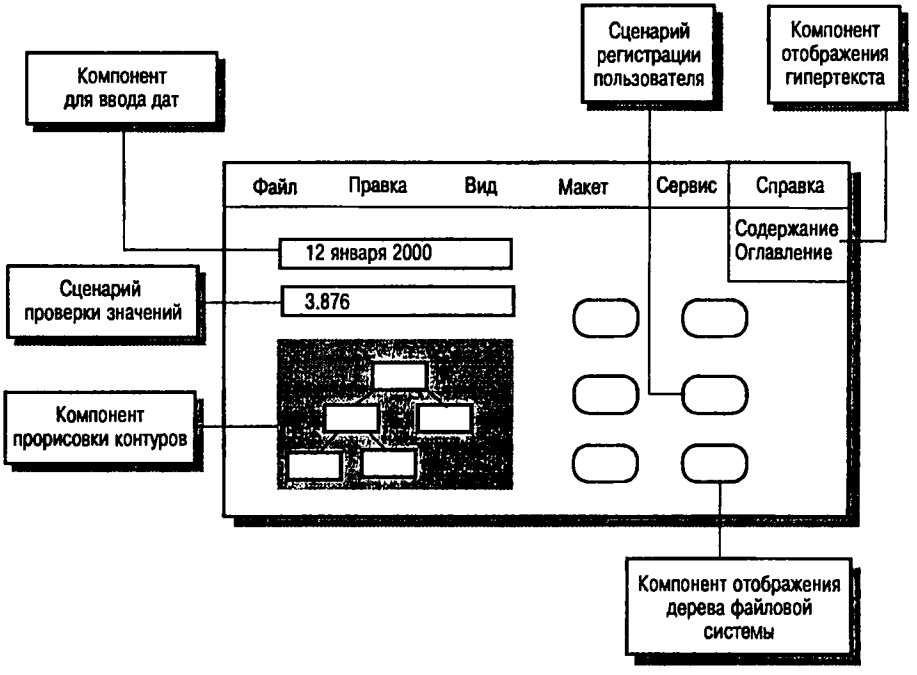


Рис. 8.9. Визуальное программирование с повторным использованием компонентов

Visual Basic – пример семейства языков, названных языками сценариев [268]. Языки сценариев – нетипичные языки высокого уровня, разработанные для интеграции компо-

нентов в единую систему. Ранним примером языка сценариев может служить оболочка Unix [56], с тех пор были созданы другие более мощные языки создания сценариев [267, 225, 337, 4*]. Эти языки имеют управляющие структуры и графические инструментальные средства, радикально уменьшающие время разработки систем.

Описанный подход к разработке систем предоставляет возможность быстро разработать относительно малые и простые приложения, которые могут быть построены одним лицом или небольшим коллективом разработчиков. Для больших систем, которые должны разрабатываться большими коллективами, подобный подход организовать более сложно, поскольку не существует явной архитектуры системы и часто имеются сложные зависимости между различными компонентами системы. В этом причина трудностей при внесении изменений в систему. Кроме того, ограничен набор компонентов, поэтому бывает трудно реализовать нестандартные пользовательские интерфейсы. Общий покомпонентный метод разработки, обсуждаемый в главе 14, является более подходящим для больших программных систем.

8.3. Прототипирование пользовательских интерфейсов

Графические интерфейсы пользователя в настоящее время являются стандартом для интерактивных систем. Усилия, вкладываемые в определение, проектирование и реализацию такого интерфейса, составляют значительную часть стоимости разработки приложения. Как отмечается в главе 15, разработчики не должны навязывать пользователям свою точку зрения на проектируемый интерфейс. Пользователи должны принимать активное участие в процессе проектирования интерфейса. Такой взгляд на разработку интерфейса привел к подходу, названному проектированием, ориентированным на пользователя (user-centred design) [258], который основан на прототипировании интерфейса и участии пользователя в процессе его проектирования.

В этом аспекте прототипирование — необходимая часть процесса проектирования пользовательского интерфейса. Из-за динамической природы пользовательских интерфейсов текстовых описаний и диаграмм недостаточно для формирования требований к интерфейсу. Поэтому эволюционное прототипирование с участием конечного пользователя — единственный приемлемый способ разработки графического интерфейса для программных систем.

Генераторы интерфейсов — это графические системы проектирования экранных форм, где интерфейсы компонуются из элементов типа меню, полей, пиктограмм и кнопок, которые, в свою очередь, можно просто выбрать из меню и поместить в экранную форму. Как уже упоминалось, системы этого типа — необходимая часть систем программирования баз данных. В книге [316] рассмотрен ряд таких систем. Генераторы интерфейсов создают хорошо структурированную программу, сгенерированную по спецификации интерфейса.

Миллионы людей сегодня имеют доступ к Web-броузерам. Они поддерживают язык разметки гипертекста HTML, который позволяет создавать пользовательские интерфейсы. Кнопки, поля, формы и таблицы могут быть включены в Web-страницы так же, как и средства мультимедиа. Сценарии обработки событий и данных, связанные с объектами интерфейса, могут выполняться или на машине Web-клиента, или на Web-сервере.

Из-за широких возможностей Web-броузеров и мощности языка HTML в настоящее время все больше пользовательских интерфейсов строятся как Web-ориентированные. Как показано в главе 26, посвященной наследуемым системам, такие интерфейсы — при-

надлежность не только новых систем; они заменяют интерфейсы, построенные на текстовых формах, в широком круге наследуемых систем.

Для Web-ориентированных интерфейсов прототипы можно создавать с помощью стандартных редакторов Web-страниц, которые, по существу, строят пользовательские интерфейсы. Объекты на Web-странице определяются, как и связанные с ними операции, с помощью встроенных средств языка HTML (например, связывание с другой страницей) или с помощью языка Java либо сценариев.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Прототип системы разрабатывается для того, чтобы дать конечным пользователям конкретное представление о возможностях системы. Прототип помогает в разработке системных требований.
- Поскольку возрастают требования к скорости разработки программного обеспечения, прототипирование становится все более используемой методикой разработки малых и среднего размера систем, особенно бизнес-систем.
- Экспериментальные прототипы используются только для формирования системных требований. В эволюционном прототипировании прототип проходит через несколько версий к окончательному варианту системы.
- При экспериментальном прототипировании сначала разрабатываются части системы, которые понятны менее всего; в эволюционном прототипировании сначала разрабатываются части системы, которые понятны лучше всего.
- В прототипировании важна быстрая разработка прототипа системы. Чтобы быстро создать прототип, не учитываются некоторые функциональные возможности системы или ослабляются нефункциональные требования, такие как быстродействие и надежность.
- Методы прототипирования включают использование языков очень высокого уровня, языков программирования баз данных и построение прототипа из пригодных для повторного использования компонентов.
- Пользовательские интерфейсы должны всегда разрабатываться с использованием прототипов, поскольку их невозможно точно определить только с помощью статической модели. Пользователи должны быть включены в процесс оценки и разработки прототипа интерфейса.

Упражнения

8.1. Исследуйте возможность прототипирования в процессе разработки программного обеспечения в вашей организации. Напишите отчет для вашего менеджера, показывая классы проектов, где должно использоваться прототипирование, и рассчитайте ожидаемые затраты и выгоды от использования прототипирования.

8.2. Объясните, почему для разработки больших систем рекомендуется экспериментальное прототипирование.

8.3. Какие особенности языков, подобных Smalltalk и Lisp, способствуют поддержке быстрого прототипирования?

8.4. В каких обстоятельствах вы рекомендовали бы прототипирование как средство обоснования системных требований?

8.5. Опишите трудности, которые могут возникнуть при прототипировании встроенных компьютерных систем реального времени.

8.6. Спроектируйте программную систему преобразования требований в формальную спецификацию. Прокомментируйте преимущества и недостатки следующих стратегий разработки такой системы.

- Разработайте экспериментальный прототип с помощью языка, подобного Smalltalk. Оцените этот прототип, затем сделайте обзор требований. Разработайте конечную систему, используя язык C.
- Разработайте систему согласно существующим требованиям, используя язык Java, и затем модифицируйте ее, чтобы адаптировать к изменениям требований пользователя.
- Разработайте систему, используя эволюционное прототипирование, с помощью языка типа Smalltalk. Измените прототип в соответствии с новыми пользовательскими запросами.

8.7. Обсудите прототипирование на основе повторного использования компонентов и опишите проблемы, которые могут при этом возникнуть. Как наиболее эффективно определить пригодные для повторного использования компоненты?

8.8. Каковы преимущества и недостатки использования механизма OLE для быстрой разработки приложений?

8.9. Благотворительная организация попросила вас создать макет системы, которая следила бы за всеми получаемыми ими пожертвованиями. Эта система должна сохранять имена и адреса жертвующих, их интересы, пожертвованную сумму и дату пожертвования. Если пожертвование достигает определенной суммы, жертвующий может добавить условия к пожертвованию (например, пожертвование должно быть израсходовано на определенный проект), система должна следить за такими пожертвованиями и за тем, как они были израсходованы. Обсудите, как использовать прототип системы, имея в виду, что системой будут пользоваться как постоянные работники благотворительной организации, так и добровольцы. Многие из добровольцев — пенсионеры, которые имеют малый опыт работы с компьютером или вовсе не имеют такого опыта.

8.10. Вы разработали экспериментальный прототип системы для заказчика, который его полностью удовлетворил. Заказчик утверждает, что нет необходимости разрабатывать конечную систему, а вы можете поставить прототип, и предлагает за это хорошую цену. Вы знаете, что в будущем могут быть проблемы с сопровождением системы. Обсудите, что вы ответите этому заказчику.

9

Формальные спецификации ПО

Цели

Цель настоящей главы — представить формальные спецификации, которые можно использовать для детализации спецификации системных требований. Прочитав эту главу, вы должны:

- понимать, почему методы формальной спецификации помогают обнаруживать проблемы в системных требованиях;
- знать, как для формирования требований, описывающих интерфейс, используются алгебраические методы формальных спецификаций;
- иметь представление о том, как формальные методы можно использовать для специфицирования поведения системы.

Содержание

Традиционные технические дисциплины, такие, как электротехника или гражданское строительство, обычно легко адаптируют лучшие математические методы. Например, в машиностроении не возникало трудностей с применением методов математического анализа. Однако инженерия программного обеспечения не идет таким путем. Хотя прошло более 25 лет исследований по использованию математических методов в процессе создания ПО, воздействие этих методов все же ограниченно. Так называемые формальные методы разработки программных систем широко не используются. Многие компании, разрабатывающие ПО, не считают экономически выгодным применение этих методов в процессе разработки.

Термин "формальные методы" подразумевает ряд операций, в состав которых входят создание формальной спецификации системы, анализ и доказательство спецификации, реализация системы на основе преобразования формальной спецификации в программы (см. главу 3) и верификация программ. Все эти действия зависят от формальной спецификации программного обеспечения. Формальная спецификация – это системная спецификация, записанная на языке, словарь, синтаксис и семантика которого определены формально. Необходимость формального определения языка предполагает, что этот язык основывается на математических концепциях. Здесь используется область математики, которая называется дискретной математикой и основывается на алгебре, теории множеств и алгебре логики.

В 1980-х годах многие исследователи считали, что формальные спецификации и формальные методы являются наиболее эффективным путем улучшения качества ПО. Они привели веские доводы в пользу того, что строгий и детальный анализ, который является неотъемлемой частью формальных методов, приведет к созданию программ с малым количеством ошибок. Они предсказывали, что к XXI столетию бо́льшая часть программного обеспечения будет разрабатываться с использованием формальных методов.

Теперь ясно, что эти предсказания не сбылись, причем по целому ряду причин.

1. *Успехи инженерии программного обеспечения.* Использование в процессе проектирования и разработки программных систем таких технологий инженерии ПО, как структурные методы, управление конфигурацией, сокрытие информации и т.д., привело к повышению качества программных продуктов. Это противоречит бытовавшим представлениям, что повышение качества программ может быть достигнуто только путем доказательства их правильности.
2. *Изменения на рынке программных продуктов.* В 1980-х годах качество являлось ключевой проблемой в создании ПО. В настоящее время главным критерием при разработке многих классов ПО является не качество, а время поставки его на рынок. Программное обеспечение должно быть разработано быстро, и клиенты готовы принять его с некоторыми дефектами, если будет обеспечена быстрая поставка. Методы быстрой разработки ПО не очень хорошо согласуются с формальной спецификацией. Конечно, качество является важным показателем, но оно должно достигаться в контексте с быстрой поставкой на рынок.
3. *Ограниченная область применения формальных методов.* Формальные методы обычно плохо подходят для описания взаимодействий пользователя и определения пользовательских интерфейсов. Пользовательские интерфейсы являются составной частью большинства современных систем, здесь польза от применения формальных методов ограниченна.
4. *Ограниченная масштабируемость формальных методов.* В успешных проектах формальные методы, как правило, использовались для разработки только относительно небольшого критического ядра системы. Проблема усугубляется недостатком средств поддержки таких методов.

Эти факторы привели к тому, что риски применения формальных методов в большинстве проектов ПО перевешивают возможные выгоды от их использования. Затраты и проблемы введения формальных методов в процесс разработки очень высоки. Однако формальная спецификация является отличиым способом обнаружения ошибок в требованиях и средством, обеспечивающим однозначность системной спецификации. Во всех успешных проектах, использовавших формальные методы, сообщалось об уменьшении количества ошибок в законченных программных системах.

Таким образом, в системах, где возможно применение формальных методов, оно может быть оправданным и, вероятно, будет рентабельным. Использование формальных методов возрастает в специальной области разработки критических систем, где очень важны такие свойства, как безопасность, безотказность и защищенность. Для таких систем, как описанные в части IV, аттестация требует больших затрат, а стоимость отказов весьма велика. В этом случае рентабельно использовать формальные методы, поскольку они могут уменьшить эти затраты.

Примерами критических систем, при разработке которых успешно применялись формальные методы, являются информационные системы управления воздушным транспортом [148], системы сигнализации на железной дороге [90], бортовые системы космических кораблей [102] и медицинские системы управления [183, 185]. Они также были использованы для специфицирования пакетов программных средств [253], части системы CICS (абонентская информационно-управляющая система) компании IBM [349] и ядра систем, работающих в режиме реального времени [324]. Метод "чистой комнаты" разработки ПО [239, 219, 284], который рассмотрен в главе 19, основывается на формальных доказательствах того, что программа соответствует своей спецификации.

9.1. Формальные спецификации в процессе разработки ПО

В главе 6 определены три уровня спецификации программного обеспечения. Это пользовательские и системные требования и спецификация структуры программной системы. Пользовательские требования наиболее обобщенные, спецификация структуры наиболее детальна. Формальные математические спецификации находятся где-то между системными требованиями и спецификацией структуры. Они не содержат деталей реализации системы, но должны представлять ее полную математическую модель.

По мере разработки спецификации участие заказчика уменьшается, а участие подрядчиков и непосредственно разработчиков ПО возрастает. На ранних стадиях разработки спецификация должна быть "ориентирована на заказчика" и написана так, чтобы он мог ее понять. Однако на заключительной стадии процесса разработки должна быть получена спецификация, в основном предназначенная для подрядчиков и разработчиков ПО, поскольку она будет служить основой для реализации системы. Эта конечная спецификация может быть формальной.

На рис. 9.1 показаны этапы разработки спецификации ПО и их взаимосвязи с процессом проектирования. Этапы разработки спецификации, показанные на рис. 9.1, не являются независимыми и не обязательно разрабатываются в приведенной последовательности. На рис. 9.2 показано, что разработка спецификация и проектирование могут выполняться параллельно, когда информация от этапов разработки спецификации передается к этапам проектирования и наоборот.

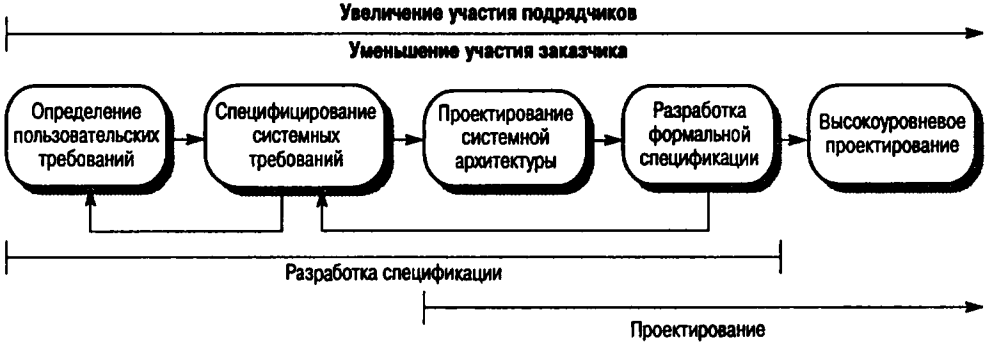


Рис. 9.1. Разработка спецификации и проектирование

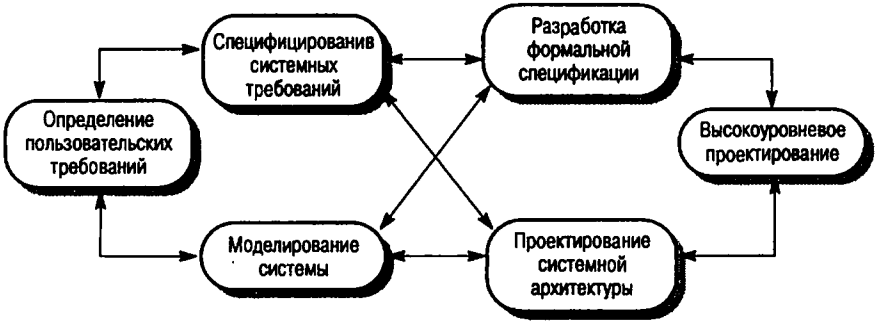


Рис. 9.2. Разработка формальной спецификации

Создание формальной спецификации требует детального анализа системы, который позволяет обнаружить ошибки и несоответствия в спецификации неформальных требований. Эта возможность обнаружения ошибок – наиболее важный аргумент для использования формальной спецификации [147]. Проблемы в требованиях, которые остаются необнаруженными до последних стадий процесса разработки ПО, обычно требуют больших затрат на исправление.

Разработка и анализ формальной спецификации требуют дополнительных затрат. На рис. 9.3 показана стоимость создания ПО при разработке формальной спецификации и без нее. При обычном процессе разработки ПО стоимость аттестации системы составляет около 50% всей стоимости разработки, а стоимость проектирования и реализации системы в два раза больше стоимости разработки спецификации. При использовании формальной спецификации стоимости разработки спецификации и реализации системы соизмеримы, а стоимость аттестации значительно снижается, поскольку в процессе разработки формальной спецификации обнаруживаются и устраняются недоработки в требованиях, тем самым исключается переделка системы на последних стадиях ее создания.

Существует два основных подхода к разработке формальной спецификации, которые используются для написания детализированных спецификаций нетривиальных программных систем.

1. Алгебраический подход, при котором система описывается в терминах операций и их отношений.

2. Подход, ориентированный на моделирование, при котором модель системы строится с использованием математических конструкций, таких, как множества и последовательности, а системные операции определяются тем, как они изменяют состояния системы.

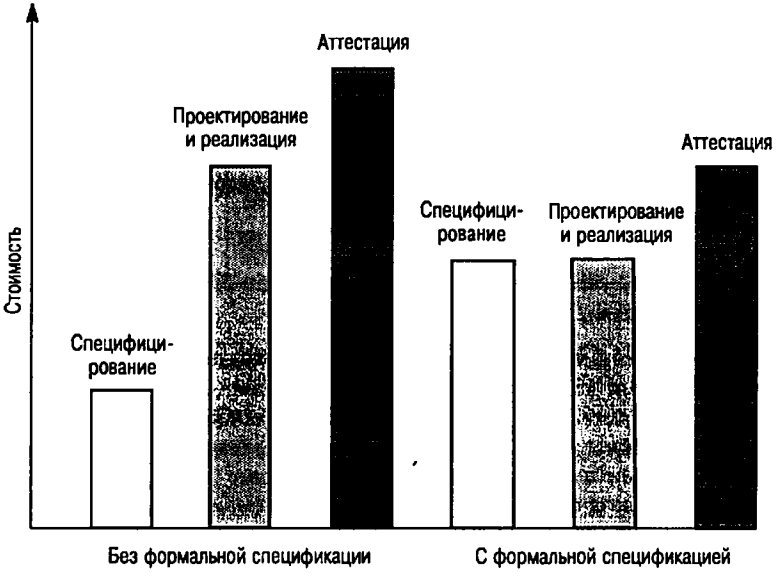


Рис. 9.3. Стоимость разработки ПО с формальной спецификацией

Для разработки формальных спецификаций последовательных и параллельных систем в настоящее время создано несколько языков, представленных в табл. 9.1. В этой главе описаны оба подхода. Приведенные далее примеры показывают, как построение формальных спецификаций приводит к точным и детализированным спецификациям, но здесь не обсуждаются языки разработки спецификаций и методы специфицирования. Вы можете получить более полное описание языков разработки формальных спецификаций на Web-странице данной книги.

Таблица 9.1. Языки разработки формальных спецификаций

Тип языка	Последовательные системы	Параллельные системы
Алгебраический	Larch [144, 145], OBJ [123]	Lotos [52]
Основанный на моделях	Z [325], VDM [192], B [343]	CSP [163], сети Петри [277]

9.2. Спецификирование интерфейсов

Большие системы обычно разбиваются на подсистемы, которые разрабатываются независимо друг от друга. Подсистемы могут использовать другие подсистемы, поэтому необходимой частью процесса спецификирования является определение интерфейсов подсистем. Если интерфейсы определены и согласованы, подсистемы можно разрабатывать независимо друг от друга.

Интерфейс подсистемы часто определяется как набор абстрактных типов данных и объектов (рис. 9.4), при этом только через интерфейс доступны описание данных и операции над ними. Поэтому спецификацию интерфейса подсистемы можно рассматривать как объединение спецификаций компонентов, что в итоге и составит описание интерфейса подсистемы.

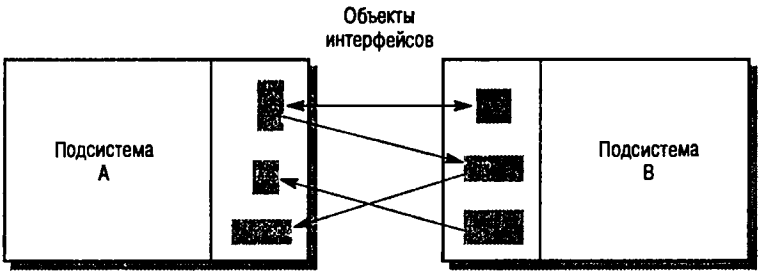


Рис. 9.4. Объекты интерфейсов подсистем

Точные спецификации интерфейсов подсистем необходимы для разработчиков, которые пишут программный код, обращающийся к сервисам других подсистем. Спецификации интерфейсов содержат информацию о том, какие сервисы доступны в других подсистемах и как получить к ним доступ. Ясный и однозначный интерфейс подсистем уменьшает вероятность ошибок во взаимоотношениях между ними.

Алгебраический подход первоначально был разработан для описания интерфейсов абстрактных типов данных, где типы данных определяются скорее спецификациями операций над данными, чем способом представления самих данных. Это очень похоже на определение классов объектов. Алгебраический подход к формальным спецификациям определяет абстрактный тип данных в терминах операций над данными.

Первый метод спецификации абстрактных типов данных описан в работе [143]. В [78] этот метод был расширен, чтобы предоставить возможность создания завершенной системной спецификации. Одновременно была разработана алгебраическая спецификация абстрактных типов данных [220]. В этой связи отметим, что, вероятно, наилучшим из известных языков алгебраической спецификации является LARCH [145].

Структура спецификации объекта показана на рис. 9.5 и состоит из четырех компонентов.

- Введение, где объявляется класс (sort) объектов. Класс — это общее название для множества объектов. Он обычно реализуется как тип данных. Введение может также включать объявление импорта (imports), где указываются имена спецификаций, определяющие другие классы. Импортирование спецификаций делает эти классы доступными для использования.
- Описательная часть, в которой неформально описываются операции, ассоциированные с классом. Это делает формальную спецификацию более простой для понимания. Формальная спецификация дополняет это описание, обеспечивая однозначный синтаксис и семантику операций.
- Часть сигнатур, в которой определяется синтаксис интерфейса объектного класса или абстрактного типа данных. Здесь описываются имена операций, количество и типы их параметров, а также классы выходных результатов операций.
- Часть аксиом, где определяется семантика операций посредством создания ряда аксиом, которые характеризуют поведение абстрактного типа данных. Эти аксио-

мы связывают операции создания объектов класса с операциями, проверяющими их значения.

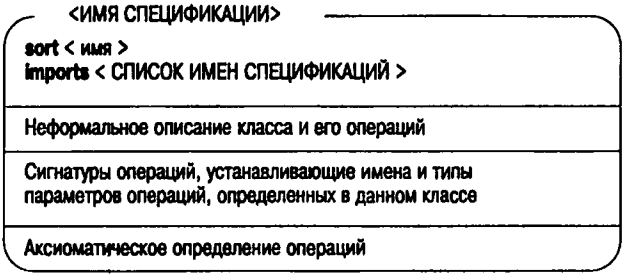


Рис. 9.5. Структура алгебраической спецификации

Процесс разработки формальной спецификации интерфейса подсистемы включает следующие действия.

1. *Структурирование спецификации.* Представление неформальной спецификации интерфейса в виде множества абстрактных типов данных или объектных классов. Также неформально определяются операции, ассоциированные с каждым классом.
2. *Именование спецификаций.* Задаются имена для каждой спецификации абстрактного типа, определяются параметры спецификаций (если они необходимы) и имена определяемых классов.
3. *Определение операций.* На основании списка выполняемых интерфейсом функций для каждой спецификации определяется связанный с ней набор операций. Необходимо предусмотреть операции по созданию экземпляров классов, по изменению значений экземпляров классов и по проверке этих значений. Вероятно, придется добавить новые функции к первоначально определенному списку функций интерфейса.
4. *Неформальная спецификация операций.* Написание неформальной спецификации для каждой операции, где должно быть указано, как операции воздействуют на определяемый класс.
5. *Определение синтаксиса операций.* Определение синтаксиса и параметров для каждой операции. Это часть сигнатуры формальной спецификации.
6. *Определение аксиом.* Определение семантики операций путем описания условий, которые должны выполняться для различных комбинаций операций.

Для пояснения методики алгебраической спецификации, рассмотрим простую структуру данных связанного списка, спецификация которого показана на рис. 9.6.

Предположим, что первый этап разработки спецификации списка, а именно структурирование спецификации, выполнен. Имя спецификации и имя класса может быть одним и тем же, хотя полезно проводить различие между ними, используя какое-либо соглашение. Например, я использую заглавные буквы для имени спецнфикации (LIST) и прописные буквы с первой заглавной буквой для имени класса (List). Поскольку списки могут содержать элементы разных типов, спецификация имеет общий параметр Elem (Элемент). Тип Elem может представлять целое число, строку, список и т.д.

LIST (Elem)

sort List
imports INTEGER

Определение списка, в котором элементы добавляются в конец, а изалекаются из вершины (начала) списка. Операция Create (Создать) создает пустой список; операция Cons (Конструирование) создает новый список, содержащий указанный элемент; Length (Длина) возвращает размер списка; Head (Верхний элемент) возвращает элемент из вершины списка; операция Tail изменяет список путем удаления из него первого элемента (элемента на вершине списка). Значения типа Elem (Элемент) не определены.

Create → List
Cons (List, Elem) → List
Head (List) → Elem
Length (List) → Integer
Tail (List) → list

Head (Create) = Undefined **exception** (пустой список)
Head (Cons (L, v)) = **if** L = Create **then** v **else** Head (L)
Length (Create) = 0
Length (Cons (L, v)) = Length (L) + 1
Tail (Create) = Create
Tail (Cons (L, v)) = **if** L = Create **then** Create **else** Cons (Tail (L), v)

Рис. 9.6. Спецификация связанного списка

В общем случае для каждого абстрактиого типа данных набор необходимых операций должен содержать операцию по созданию нового экземпляра этого типа данных и операцию по конструированию экземпляра данного типа из имеющихся элементов (в нашем примере это операции Create и Cons). В случае применения списков также должны быть операции для получения значения первого элемента списка (в нашем примере операция Head), операция, которая изменяет список путем удаления его первого элемента (Tail), и операция подсчета количества элементов в списке (Length).

При определении синтаксиса этих операций следует установить, какие для них необходимы параметры и каковы должны быть результаты операций. В общем случае входные параметры принадлежат или определяемому классу (в данном случае класс List) или более общему (родовому) классу. Результаты операций также принадлежат тем же классам или некоторому другому, например Integer или Boolean; операция Length возвращает целое число. Декларация импорта, объявляющая использование спецификации целых чисел, должна быть включена в спецификацию. Аксиомы, определяющие семантику абстрактного типа данных, написаны с использованием ранее определенных операций.

Операции над абстрактным типом данных обычно относятся к одному из двух классов.

1. *Операции конструирования*, которые создают или изменяют объекты класса. Обычно их называют Create (Создать), Update (Изменить), Add (Добавить) или, как в нашем случае, Cons (Конструирование).
2. *Операции проверки*, которые возвращают атрибуты класса. Обычно им дают имена, соответствующие именам атрибута, или имена, подобные Eval (Значение), Get (Получить) и т.п.

Хорошим эмпирическим правилом для написания алгебраической спецификации является создание аксиом для каждой операции конструирования с применением всех операций проверки. Это означает, что если есть m операций конструирования и n операций проверки, то должно быть определено m × n аксиом.

Операции конструирования, связанные с абстрактным типом данных, часто очень сложны и могут определяться через другие операции конструирования и проверки. Если операции конструирования определены посредством других операций, то необходимо определить операции проверки, используя более примитивные конструкции.

В спецификации списка операциями конструирования являются Create, Cons и Tail, которые создают списки. Операциями проверки являются Head и Length, которые используются для получения значений атрибутов списка. Операция Tail не является примитивной конструкцией, поэтому для нее можно не определять аксиомы с использованием операций Head и Length, но в таком случае Tail необходимо определить посредством примитивных конструкций.

При написании алгебраических спецификаций часто используется рекурсия. Результат операции Tail — список, сформированный из входного списка путем удаления верхнего элемента. Это определение подсказывает, как использовать рекурсию для построения данной операции. Операция определяется на пустых списках, затем рекурсивно переходит на непустые списки и завершается, когда результатом снова будет пустой список.

Иногда проще понять рекурсивные преобразования, используя короткий пример. Предположим, что есть список [5, 7], где элемент 5 — начало (вершина) списка, а элемент 7 — конец списка. Операция Cons([5, 7], 9) должна возвратить список [5, 7, 9], а операция Tail, примененная к этому списку, должна возвратить список [7, 9]. Приведем последовательность рекурсивных преобразований, приводящую к этому результату.

```
Tail ([5, 7, 9]) =
    = Tail (Cons ([5, 7], 9)) =
    = Cons (Tail ([5, 7]), 9) =
    = Cons (Tail (Cons ([5], 7)), 9) =
    = Cons (Cons (Tail ([5]), 7), 9) =
            = Cons (Cons (Tail (Cons ([], 5)), 7), 9) =
            = Cons (Cons ([Create], 7), 9) =
    = Cons ([7], 9) =
            = [7, 9]
```

Здесь систематически использовались аксиомы для Tail, что привело к ожидаемому результату. Аксиому для операции Head можно проверить подобным способом.

Теперь рассмотрим, как эту методику можно использовать при разработке спецификации критической системы. Предположим, что имеется система управления воздушным движением, целью которой является контроль за определенным сектором воздушного пространства. В каждом контролируемом секторе может находиться несколько самолетов, имеющих различные идентификаторы. Из соображений безопасности все самолеты должны быть разведены по высоте по крайней мере на 300 метров. В случае попытки каким-либо самолетом нарушить это ограничение, система должна выдать предупреждающий сигнал.

SECTOR

sort Sector
imports INTEGER, BOOLEAN

Enter добавляет самолет в сектор, если позволяют условия безопасности
Leave удаляет самолет из сектора
Move перемещает самолет с одной высоты на другую, если позволяют условия безопасности
Lookup определяет высоту самолета

Create создает пустой сектор
Put добавляет самолет в сектор без проверки условий безопасности
In-space проверяет, есть ли самолет в секторе
Occupied проверяет, доступна ли указанная высота

Enter (Sector, Call-sign, Height) → Sector
Leave (Sector, Call-sign) → Sector
Move (Sector, Call-sign, Height) → Sector
Lookup (Sector, Call-sign) → Height

Create → Sector
Put (Sector, Call-sign, Height) → Sector
In-space (Sector, Call-sign) → Boolean
Occupied (Sector, Height) → Boolean

```
Enter (S, CS, H) =
   if In-space (S, CS) then S exception (Семолет уже в секторе)
   elsif Occupied (S, H) then S exception (Высота занята)
   else Put (S, CS, H)

Leave (Create, CS) = Create exception (Самолет не в секторе)
Leave (Put (S, CS), H1) CS)=
   if CS = CS1 then S else Put (Leave (S, CS), CS1, H1)

Move (S, CS, H) =
    if S = Create then Create exception (В секторе нет самолетов)
   elsif not In-space (S, CS) then S exception (Самолет не в секторе)
   elsif Occupied (S, H) then S exception (Высота занята)
   else Put (Leave (S, CS), CS, H)

-- NO-HEIGHT - константа, показывающая, что значение высоты не возвращено

Lookup (Create, CS) = NO-HEIGHT exception (Самолет не в секторе)
Lookup (Put (S, CS1, H1), CS) =
   if CS =CS1 then H1 else Lookup (S, CS)

Occupied (Create, H) = false
Occupied (Put (S, CS1, H1), H)=
   if (H1 > H and H1 - H ≤ 300) or (H > H1 and H - H1 ≤ 300) then true
   else Occupied (S, H)
In-space (Create, CS) = false
In-space (Put (S, CS1, H1), CS) =
   if CS = CS! then true else In-space (S, CS)
```

Рис. 9.7. Спецификация класса Sector, представляющего сектор контролируемого воздушного пространства

Чтобы упростить описание, я определил только ограниченное число операций над объектом-сектором. В реальной системе, конечно, будет намного больше операций и более сложными будут условия безопасности полета самолетов. Основные операции следующие.

1. Enter (Ввод). Добавляет самолет (представляемый идентификатором) в воздушное пространство на указанной высоте. На этой высоте или на расстоянии ближе 300 метров от него не должно быть другого самолета.
2. Leave (Выход). Удаляет указанный самолет из контролируемого сектора. Она применяется, если самолет перемещается в соседний сектор.
3. Move (Перемещение). Перемещает самолет с одной высоты на другую. Опять проверяются условия безопасности — расстояние между самолетами должно быть не менее 300 метров.
4. Lookup (Просмотр). Определяет текущую высоту самолета в секторе.

Определение этих операций упростится, если определены также другие операции.

1. Create (Создать). Стандартная операция для любого абстрактного типа даиных. Она создает пустой экземпляр данного типа. В нашем случае эта операция задает сектор, в котором отсутствуют самолеты.
2. Put (Поместить). Более простая версия операции Enter. Она добавляет новый самолет в сектор без проверки ограничений.
3. In-space (Проверка пространства). Возвращает значение истины, если указанный самолет находится в контролируемом секторе, в противном случае ее значение ложно.
4. Occupied (Занятый). Возвращает значение истины, если внутри 300-метровой зоны по высоте есть самолет, в противном случае ее значение ложно.

Простые операции определяются для того, чтобы впоследствии использовать их как блоки для компоиовки более сложных операций. Алгебраическая спецификация класса Sector (Сектор) показана на рис. 9.7.

По существу, основными операциями конструирования являются Create и Put, которые использованы в спецификациях других операций. Occupied и In-space являются операциями проверки, которые определяют использование операций Create и Put. Как выполняются эти и другие операции, легко понять из спецификации.

9.3. Спецификация поведения систем

Простые алгебраические методы, описанные в предыдущем разделе, подходят для описанния интерфейсов, когда операции, ассоциированные с объектом, не зависят от состояния объекта. Тогда результаты любой операции не зависят от результатов предыдущих операций. Если это условие не выполняется, алгебраические методы могут стать громоздкими. Более того, я думаю, что алгебраические описания поведения систем часто искусственны и трудны для понимания.

Альтернативным подходом к созданию формальных спецификаций, который широко используется в программных проектах, является спецификация, основанная на моделях системы. Такие спецификации используют модели состояний системы. Системные операции определяются посредством изменений состояний системной модели. Таким образом определяется поведение системы.

Для разработки спецификаций, основанных на системных моделях, используются системы нотаций методов VDM [192, 193], B [348] и Z [158, 325]. Здесь я использую нотацию метода Z. В этом методе система описывается на основе теории множеств, которая дополняется специальными логическими структурами, разработанными для специфицирования программных систем. Полное описание метода Z очень объемно. Поэтому я представлю несколько небольших примеров для иллюстрации метода и приведу необходимые обозначения. Полное описание метода Z дается в [98, 347, 84].

Формальные спецификации могут быть трудными для чтения и громоздкими, особенно если используются большие математические формулы. Это замедляет разработку ПО. Разработчики метода Z обратили внимание на эту проблему. В этом методе спецификации

представляются как неформальный текст, дополненный формальными описаниями. Формальная часть спецификации состоит из небольших простых описаний (называемых схемами), которые визуально отделяются от остального текста спецификации (рис. 9.8). Схемы используются для введения переменных состояний и определения ограничений и операций над состояниями. В методе также предусмотрены операции, выполняемые над схемами, в частности для построения, переименования и сокрытия схем.

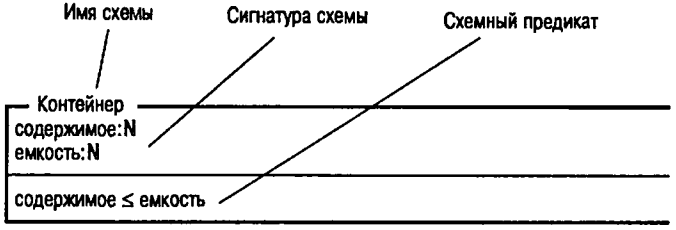


Рис. 9.8. Структура Z-схемы

Сигнатура схемы определяет сущности, которые составляют состояние системы, схемные предикаты – это набор условий, которые должны быть всегда истинны для этих сущностей. Если схема определяет операции, предикаты могут представлять пред- и постусловия для этих операций.

Для иллюстрации применения метода Z в разработке спецификации критической системы рассмотрим упрощенный пример системы инсулинового насоса, используемой диабетиками. У диабетиков нарушен естественный метаболизм сахара, им требуются инъекции инсулина – гормона, необходимого для метаболизма глюкозы. Данная система контролирует уровень сахара в крови больного и, если требуется, производит автоматическую инъекцию инсулина. Этот пример также использован в части IV, где рассматривается разработка критических систем.

Уровень сахара в крови больного проверяется через равные промежутки времени, и, если он увеличивается, производится инъекция инсулина, которая понижает уровень сахара. На рис. 9.9 показана структура инсулинового насоса.

1. *Набор игл.* Подсоединены к насосу, используются для инъекции инсулина в тело диабетика.
2. *Датчик.* Измеряет уровень сахара в крови больного. В формальной спецификации операция получения данных от датчика названа reading?.
3. *Насос.* Передает инсулин из резервуара к набору игл. В формальной спецификации величина дозы инсулина названа dose.
4. *Управляющее устройство.* Управляет объектами системы. Имеет переключатель "Включено – Выключено", кнопку отмены и кнопку установки дозы инсулина. Для упрощения формальной спецификации эти элементы управления не описаны.
5. *Устройство аварийной сигнализации.* Подает звуковой сигнал при возникновении проблем. В спецификации сигнал на входе устройства аварийной сигнализации обозначен как alarm!.
6. *Индикаторы.* Имеется два индикатора: один показывает последний анализ уровня сахара в крови, другой – информацию, выдаваемую системой пользователю. Эти индикаторы в формальной спецификации обозначаются display1! и display2!.
7. *Часы.* Обеспечивают управляющее устройство значением текущего времени.

Даже для небольшой системы, подобной системе управления инъекциями инсулина, формальная спецификация довольно обширна. Хотя основные системные операции просты, существует много аварийных ситуаций, которые необходимо учитывать. Здесь приведены только некоторые из основных Z-схем спецификации и объяснено, что они означают.

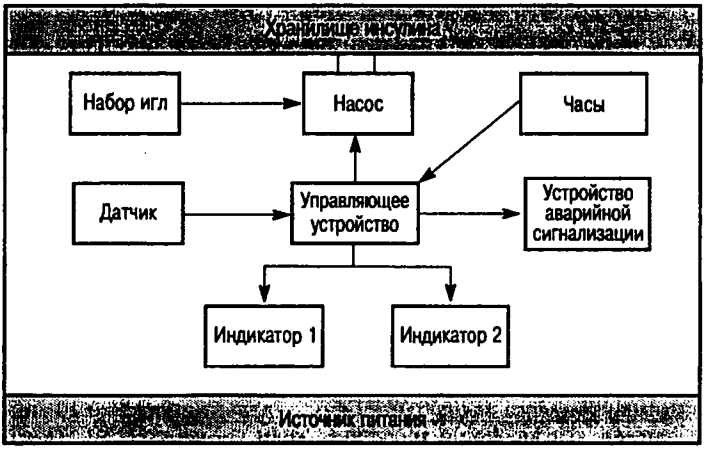


Рис. 9.9. Блок-схема инсулинового насоса

Основная схема Insulin-pump (инсулиновый насос), которая моделирует состояния инсулинового насоса, показана на рис. 9.10. Схема разбита на две части. В верхней части объявляются имена и типы, в нижней приведены условия, которые должны всегда выполняться.

```
Insulin_pump
reading?: N
dose, cumulative_dose: N
r0, r1, r2:N        // используются для записи 3 последних анализа
capacity: N
alarm!: (off, on)
pump!: N
display1!, display2!: STRING
---
dose ≤ capacity ∧ dose ≤ 5 ∧ cumulative_dose ≤ 50
capacity ≥ 40 ⇒ display1! = " "
capacity ≤ 39 ∧ capacity ≥ 10 ⇒ display1! = "Мало инсулина"
capacity ≤ 9 ⇒ alarm! = on ∧ display1! = "Очень мало инсулина"
r2 = reading?
```

Рис. 9.10. Z-схема инсулинового насоса

Состояние системы моделируется посредством ряда переменных. В соответствии с соглашениями, принятыми в методе Z, имена, заканчивающиеся знаком ?, используются для представления входных данных, а имена, заканчивающиеся знаком !, представляют выходные данные. В схеме инсулинового насоса объявлены следующие имена.

1. reading?. Это неотрицательное целое число, которое представляет данные от датчика, определяющего количество сахара в крови. Это входная величина.
2. dose, cumulative_dose. Это также натуральные числа, представляющие соответственно дозу инсулина и суммарную дозу инсулина, введенную за определенный период времени.
3. r0, r1, r2. Представляют последние три значения, полученные от датчика, и используются для вычисления изменения сахара в крови.
4. capacity. Натуральное число, представляющее объем инсулина в хранилище (резервуаре).
5. alarm!. Эта выходная величина сообщает об аварийных ситуациях в системе.
6. pump!. Это натуральное число, представляющее управляющие сигналы, посылаемые к физическому насосу.

7. display1!, display2!. Эти выходные величины строкового типа представляют два текстовых индикатора. Один индикатор используется для отображения текстовых сообщений, другой — для показа введенной дозы инсулина.

Схемные предикаты определяют ряд условий, которые всегда должны быть истинными.

1. Доза должна быть меньше или равна количеству инсулина в резервуаре.
2. Однократная доза не должна превышать 5 единиц инсулина, а общая доза, введенная за определенный промежуток времени, — 50 единиц инсулина. Это условия безопасности, которые рассматриваются в главе 17.
3. display1!. Показывает сообщения о состоянии инсулинового резервуара. Если резервуар содержит 40 или больше единиц инсулина, сообщений нет. Когда в резервуаре инсулина от 10 до 40 единиц, на дисплее отображается предупреждение, если же в резервуара меньше 10 единиц, звучит звуковой сигнал и отображается соответствующее предупреждение.

Работа инсулинового насоса заключается в определении каждые 10 минут количества сахара в крови пациента и введении инсулина, если уровень сахара увеличивается (это очень упрощенное описание). Вводимое количество инсулина вычисляется согласно схеме DOSAGE (Дозировка), которая приведена на рис. 9.11. Из этой схемы видно, что на вводимую дозу влияет множество различных факторов.

```
DOSAGE
ΔInsulin_Pump
─────────────────────────────────────
(
dose = 0 ∧
  (
        ((r1 ≥ r0) ∧ (r2 = r1)) ∨
        ((r1 > r0) ∧ (r2 ≤ r1)) ∨
        ((r1 < r0) ∧ ((r1 - r2) > (r0 - r1)))
  ) ∨
dose = 4 ∧
  (
        ((r1 ≤ r0) ∧ (r2 = r1)) ∨
        ((r1 < r0) ∧ ((r1 - r2) ≤ (r0 - r1)))
  ) ∨
dose = (r2 - r1) * 4 ∧
  (
        ((r1 ≤ r0)  (r2 > r1)) ∨
        ((r1 > r0)  ((r2 - r1) ≥ (r1 - r0)))
  )
)
capacity' = capacity - dose
cumulative_dose' = cumulative_dose + dose
r0' = r1 ∧ r1' = r2
```

Рис. 9.11. Z-схема вычисления дозы

На рис. 9.11 показано часто используемое средство метода Z, а именно дельта-схема. Если имя схемы включено в раздел описаний, то это равносильно включению всех имен данной схемы, а ее условия включаются в предикатную часть. В данном случае в схему DOSAGE включаются имена и условия схемы Insulin_Pump. Если имени схемы предшествует символ Δ, то вводится новый набор величин, имена которых совпадают с именами данной схемы, но к ним добавляется символ ' (апостроф). Так обозначаются значения переменных состояний, измененные после выполнения операции. Например, если операция изменяет значение переменной val, то после операции эта переменная будет обозначаться val'. Дельта-схема DOSAGE вводит имена capacity', cumulative_dose' и т.д.

Схемы, моделирующие выходные данные инсулинового насоса, показаны на рис. 9.12. Это модели индикаторов и устройства аварийной сигнализации. Здесь также используется дельта-схема. Из схемы DISPLAY видно, что индикатор display2! показывает вычисленную дозу (Nat_to_string – функция преобразования), индикатор display1! отображает или предупреждающее сообщение, или OK. В схеме ALARM показаны условия, когда активизируется аварийная сигнализация. Она включается, если уровень сахара в крови очень низкий (меньше 3) или слишком высокий (больше 30).

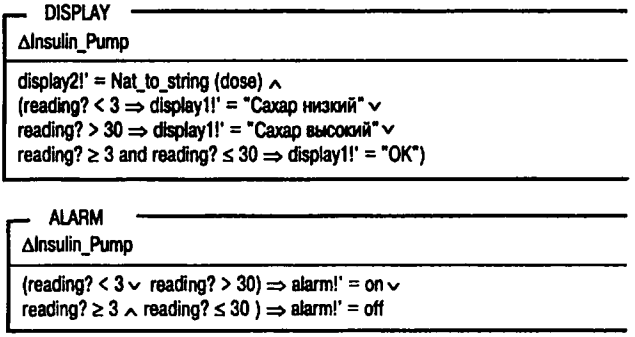


Рис. 9.12. Схемы выходных данных

Предикаты во всех Z-схемах должны быть согласованы, т.е. не должно быть условий в одной схеме, которые противоречат предикатам в другой схеме. Если в спецификации замечены противоречия, можно применить различные математические методы анализа Z-спецификации. Здесь я не буду их описывать. Рассматривая представленные четыре Z-схемы, вы могли заметить противоречия, которые я сознательно ввел, но которые могли бы появиться в реальной спецификации. В общей схеме инсулинового насоса Insulin_Pump установлено, что индикатор display1! должен показывать состояние резервуара инсулина. Однако в схеме DISPLAY этот индикатор должен показывать уровень сахара в крови. Здесь противоречие, которое следует разрешить при обсуждении системы с медицинскими экспертами и потенциальными пользователями.

Я не делал модели поведения системы во времени (в действительности нужно учитывать, что датчик контролирует уровень сахара в крови каждые 10 минут). Хотя это, конечно, возможно, но довольно громоздко; по-моему, неформальное описание действий более кратко и понятно, чем формальная спецификация.

Основным преимуществом применения формальной спецификации является возможность выявления проблем в системных требованиях. В неформальной спецификации легко упустить эти проблемы, которые все равно будут решены, но на более поздней стадии процесса разработки ПО.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Методы формальной системной спецификации дополняют методы неформальной спецификации. Они могут использоваться для детализации спецификации неформальных системных требований. Формальные спецификации являются мостом между системными требованиями и системной архитектурой.
- Формальные спецификации точны и однозначны. Они устраняют "темные" области в спецификации и проблемы неоднозначного толкования требований. Вместе с тем формальные спецификации трудны для понимания неспециалистами.

- Основным преимуществом формальных методов является то, что анализ системных требований проводится на ранней стадии разработки ПО. Исправление ошибок на этой стадии дешевле, чем внесение изменений в законченную систему.
- Методы формальной спецификации наиболее подходят для разработки критических систем, где безопасность, безотказность и защищенность системы особенно важны. Они могут также использоваться для разработки стандартов.
- Алгебраические методы формальных спецификаций особенно подходят для разработки интерфейсов, когда интерфейс определен как набор классов объектов или абстрактных типов данных. Эти методы скрывают состояние системы и определяют ее в терминах отношений между интерфейсными операциями.
- Методы формальной спецификации, основанные на системных моделях, используют математические конструкции теории множеств. Операции определяются по способу воздействия на состояния системы. Это упрощает создание спецификаций поведения системы.

Упражнения

9.1. Объясните, почему архитектурное проектирование системы должно предшествовать разработке формальной спецификации.

9.2. Перед вами поставлена задача "продажи" методов формальной спецификации организации, разрабатывающей программное обеспечение. Как вы будете объяснять преимущества формальной спецификации скептически настроенным разработчикам ПО?

9.3. Объясните, почему необходимо определять интерфейсы подсистем как можно точнее и почему алгебраическая спецификация наиболее подходит для специфицирования интерфейсов подсистем.

9.4. Абстрактный тип данных, представляющий стек, имеет следующие операции:

New (Создать) — создает пустой стек;

Push (Добавить) — добавляет элемент в вершину стека;

Top (Вершина) — возвращает элемент на вершине стека;

Retract (Удалить) — удаляет элемент из вершины стека и возвращает модифицированный стек;

Empty (Пустой) — возвращает значение истины, если стек пустой.

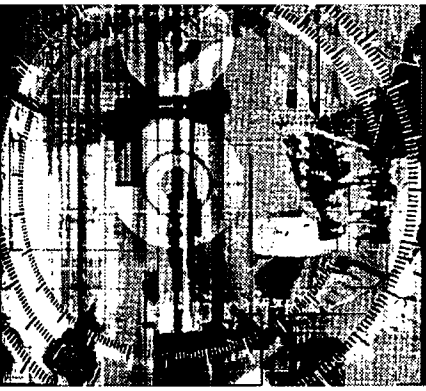
Определите этот абстрактный тип данных, используя алгебраическую спецификацию.

9.5. В примере управления сектором воздушного пространства условием безопасности является то, что два самолета не могут находиться внутри 300-метровой зоны по высоте в одном секторе. Измените спецификацию, показанную на рис. 9.7, позволяющую самолетам занимать одинаковую высоту, если расстояние по горизонтали между ними не менее 8 км. Можно игнорировать самолеты в соседних секторах.

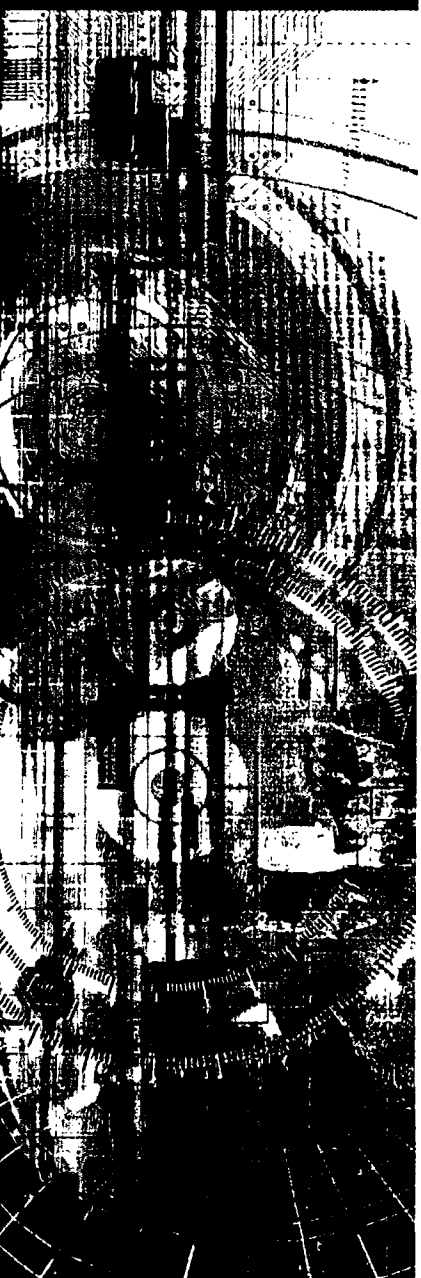
9.6. Банковские автоматы используют информацию с карточки клиента, предоставляющую идентификатор банка, номер счета и персональный идентификатор клиента. Они также получают информацию из центральной базы данных и вносят в нее изменения по завершении транзакции. Используя ваши знания работы банкоматов, напишите Z-схему, определяющую состояния системы, проверку карточки клиента и снятие денежных средств.

9.7. Измените схему инсулинового насоса, показанную на рис. 9.10, и схему DOSAGE, показанную на рис. 9.11, таким образом, чтобы пользователь системы мог отменить вычисленную дозу и сам определить дозу, необходимую для инъекции. Условия безопасности, указанные в схемах, должны сохраниться. Тот факт, что система работает в режиме отмены, должен отобразиться на индикаторе.

9.8. Вы системный инженер и вас просят назвать наилучший способ разработки программного обеспечения для сердечного стимулятора, критического по обеспечению безопасности. Вы предлагаете разработать формальную спецификацию системы, но ваше предложение отвергнуто менеджером. Вы считаете, что его доводы не обоснованы и базируются на предубеждениях. Будет ли этичной разработка системы с использованием методов, которые вы считаете неподходящими?



Проектирование



10

Архитектурное проектирование

Цели

Цель настоящей главы – познакомить читателя с архитектурой программного обеспечения и концепциями архитектурного проектирования. Прочитав эту главу, вы должны:

- понимать, для чего необходимо архитектурное проектирование ПО;
- знать различные модели, используемые при документировании системной архитектуры;
- иметь представление о разных типах архитектур ПО: структурной модели системы, модели системной декомпозиции по управлению и модели модульной декомпозиции;
- знать модели проблемно-зависимых архитектур, которые используются как основа для архитектур специализированных программных систем и как эталон при сравнении различных архитектур.

Содержание

Большие системы всегда можно разбить на подсистемы, предоставляющие связанные наборы сервисов. *Архитектурным проектированием* называют первый этап процесса проектирования, на котором определяются подсистемы, а также структура управления и взаимодействия подсистем. Целью архитектурного проектирования является описание *архитектуры программного обеспечения.*

В разделе 3.4 рассматривалась общая структура процесса проектирования. На рис. 3.9 была представлена модель процесса проектирования, первым этапом которого является архитектурное проектирование, служащее соединяющим звеном между процессом проектирования и процессом разработки требований к создаваемой системе. В идеале в спецификации требований не должно быть информации о структуре системы. В действительности же это справедливо только для небольших систем. Архитектурная декомпозиция системы необходима для структуризации и организации системной спецификации. Хорошим примером тому может служить изображенная на рис. 2.3 система управления воздушными полетами. Модель системной архитектуры часто является отправной точкой для создания спецификации различных частей системы. В процессе архитектурного проектирования разрабатывается базовая структура системы, т.е. определяются основные компоненты системы и взаимодействия между ними.

Существуют различные подходы к процессу архитектурного проектирования, которые зависят от профессионального опыта, а также мастерства и интуиции разработчиков. И все же можно выделить несколько этапов, общих для всех процессов архитектурного проектирования.

1. *Структурирование системы.* Программная система структурируется в виде совокупности относительно независимых подсистем. Также определяются взаимодействия между подсистемами. Этот этап рассматривается в разделе 10.1.
2. *Моделирование управления.* Разрабатывается базовая модель управления взаимоотношениями между частями системы. Этот этап рассматривается в разделе 10.2.
3. *Модульная декомпозиция.* Каждая определенная на первом этапе подсистема разбивается на отдельные модули. Здесь определяются типы модулей и типы их взаимосвязей. Этот этап рассматривается в разделе 10.3.

Как правило, эти этапы перемежаются и накладываются друг на друга. Этапы повторяются для все более детальной проработки архитектуры до тех пор, пока архитектурный проект не будет удовлетворять системным требованиям.

Четких различий между подсистемами и модулями нет, но, думаю, будут полезными следующие определения.

1. *Подсистема* — это система (т.е. удовлетворяет "классическому" определению "система"), операции (методы) которой не зависят от сервисов, предоставляемых другими подсистемами. Подсистемы состоят из модулей и имеют определенные интерфейсы, с помощью которых взаимодействуют с другими подсистемами.
2. *Модуль* — это обычно компонент системы, который предоставляет один или несколько сервисов для других модулей. Модуль может использовать сервисы, поддерживаемые другими модулями. Как правило, модуль никогда не рассматривается как независимая система. Модули обычно состоят из ряда других, более простых компонентов.

Результатом процесса архитектурного проектирования является документ, отображающий архитектуру системы. Он состоит из набора графических схем представлений моделей системы с соответствующим описанием. В описании должно быть указано, из ка-

ких подсистем состоит система и из каких модулей слагается каждая подсистема. Графические схемы моделей системы позволяют взглянуть на архитектуру с разных сторон. Как правило, разрабатывается четыре архитектурные модели.

1. Статическая структурная модель, в которой представлены подсистемы или компоненты, разрабатываемые в дальнейшем независимо.
2. Динамическая модель процессов, в которой представлена организация процессов во время работы системы.
3. Интерфейсная модель, которая определяет сервисы, предоставляемые каждой подсистемой через общий интерфейс.
4. Модели отношений, в которых показаны взаимоотношения между частями системы, например поток данных между подсистемами.

Ряд исследователей при описании архитектуры систем предлагают использовать специальные языки описания архитектур. В книге [29] рассматриваются основные свойства этих языков. В них основными архитектурными элементами являются компоненты и коннекторы (объединяющие звенья); эти языки также предлагают принципы и правила построения архитектур. Однако, как и другие специализированные языки, они имеют один недостаток, а именно: все они понятны только освоившим их специалистам и почти не используются на практике. Фактически использование языков описания архитектур только усложняет анализ систем. Поэтому я считаю, что для описания архитектур лучше использовать неформальные модели и системы нотации, подобные предлагаемой, например унифицированный язык моделирования UML.

Архитектура системы может строиться в соответствии с определенной архитектурной моделью [126]. Очень важно знать эти модели, их недостатки, преимущества и возможности применения. В этой главе рассматриваются структурные модели, модели управления и декомпозиции.

Вместе с тем архитектуру больших систем невозможно описать с помощью какой-либо одной модели. При разработке отдельных частей больших систем можно использовать разные архитектурные модели. Но в этом случае архитектура системы может оказаться слишком сложной, поскольку будет построена на комбинации различных архитектурных моделей. Разработчик должен подобрать наиболее подходящую модель, затем модифицировать ее соответственно требованиям разрабатываемого ПО. В разделе 10.4 рассматривается пример архитектуры компилятора, базирующейся на комбинации модели репозитория и модели потоков данных.

Архитектура системы влияет на производительность, надежность, удобство сопровождения и другие характеристики системы. Поэтому модели архитектуры, выбранные для данной системы, могут зависеть от нефункциональных системных требований.

1. *Производительность.* Если критическим требованием является производительность системы, следует разработать такую архитектуру, чтобы за все критические операции отвечало как можно меньше подсистем с максимально малым взаимодействием между ними. Чтобы уменьшить взаимодействие между компонентами, лучше использовать крупномодульные компоненты, а не мелкие структурные элементы.
2. *Защищенность.* В этом случае архитектура должна иметь многоуровневую структуру, в которой наиболее критические системные элементы защищены на внутренних уровнях, а проверка безопасности этих уровней осуществляется на более высоком уровне.

3. *Безопасность.* В этом случае архитектуру следует спроектировать так, чтобы за все операции, влияющие на безопасность системы, отвечало как можно меньше подсистем. Такой подход позволяет снизить стоимость разработки и решает проблему проверки надежности.
4. *Надежность.* В этом случае следует разработать архитектуру с включением избыточных компонентов, чтобы можно было заменять и обновлять их, не прерывая работу системы. Архитектуры отказоустойчивых систем с высокой работоспособностью рассматриваются в главе 18.
5. *Удобство сопровождения.* В этом случае архитектуру системы следует проектировать на уровне мелких структурных компонентов, которые можно легко изменять. Программы, создающие данные, должны быть отделены от программ, использующих эти данные. Следует также избегать структуры совместного использования данных.

Очевидно, что некоторые из перечисленных архитектур противоречат друг другу. Например, для того чтобы повысить производительность, необходимо использовать крупномодульные компоненты, в то же время сопровождение системы намного упрощается, если она состоит из мелких структурных компонентов. Если необходимо учесть оба требования, следует искать компромиссное решение. Ранее уже было сказано, что один из способов решения подобных проблем состоит в применении различных архитектурных моделей для разных частей системы.

10.1. Структурирование системы

На первом этапе процесса проектирования архитектуры система разбивается на несколько взаимодействующих подсистем. На самом абстрактном уровне архитектуру системы можно изобразить графически с помощью блок-схемы, в которой отдельные подсистемы представлены отдельными блоками. Если подсистему также можно разбить на несколько частей, на диаграмме эти части изображаются прямоугольниками внутри больших блоков. Потоки данных и/или потоки управления между подсистемами обозначается стрелками. Такая блок-схема дает общее представление о структуре системы.

На рис. 10.1 представлена структурная модель архитектуры для системы управления автоматической упаковкой различных типов объектов. Она состоит из нескольких частей. Подсистема наблюдения изучает объекты на конвейере, определяет тип объекта и выбирает для него соответствующий тип упаковки. Затем объекты снимаются с конвейера, упаковываются и помещаются на другой конвейер. Примеры других архитектур приведены на рис. 2.2 и 2.3.

Бэсс (Bass, [29]) считает, что подобные блок-схемы являются бесполезными представлениями системной архитектуры, поскольку из них нельзя ничего узиать ни о природе взаимоотношений между компонентами системы, ни об их свойствах. С точки зрения разработчика программного обеспечения, это абсолютно верно. Однако такие модели оказываются эффективными на этапе предварительного проектирования системы. Эта модель не перегружена деталями, с ее помощью удобно представить структуру системы. В структурной модели определены все основные подсистемы, которые можно разрабатывать независимо от остальных подсистем, следовательно, руководитель проекта может распределить разработку этих подсистем между различными исполнителями. Конечно, для представления архитектуры используются не только блок-схемы, однако подобное представление системы не менее полезно, чем другие архитектурные модели.

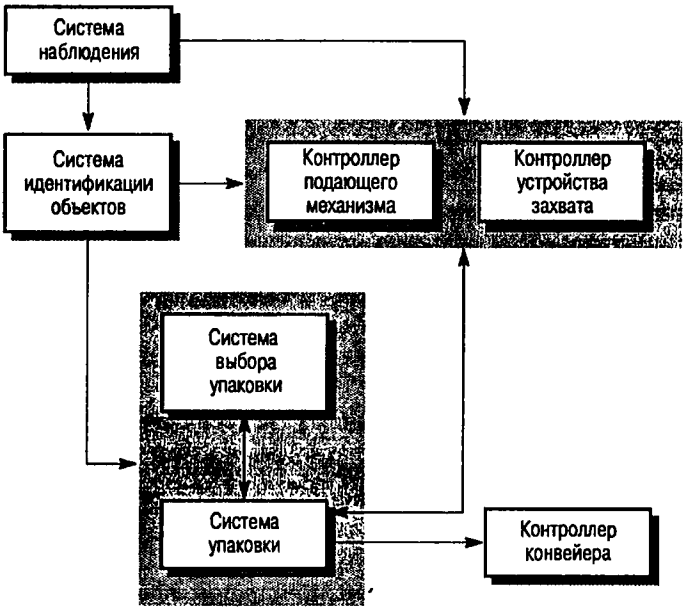


Рис. 10.1. Блок-схема системы управления автоматической упаковкой

Конечно, можно разрабатывать более детализированные модели структуры, в которых было бы показано, как именно подсистемы разделяют данные и как взаимодействуют друг с другом. В этом разделе рассматриваются три стандартные модели, а именно: модель репозитория, модель клиент/сервер и модель абстрактной машины.

10.1.1. Модель репозитория

Для того чтобы подсистемы, составляющие систему, работали эффективнее, между ними должен идти обмен информацией. Обмен можно организовать двумя способами.

1. Все совместно используемые данные хранятся в центральной базе данных, доступной всем подсистемам. Модель системы, основанная на совместном использовании базы данных, часто называют *моделью репозитория*.
2. Каждая подсистема имеет собственную базу данных. Взаимообмен данными между подсистемами происходит посредством передачи сообщений.

Большинство систем, обрабатывающих большие объемы данных, организованы вокруг совместно используемой базы данных, или репозитория. Поэтому такая модель подойдет к приложениям, в которых данные создаются в одной подсистеме, а используются в другой. Примерами могут служить системы управления информацией, системы автоматического проектирования и CASE-средства.

На рис. 10.2 представлен пример архитектуры интегрированного набора CASE-инструментов, основанный на совместно используемом репозитории. Считается, что для CASE-средств первый совместно используемый репозиторий был разработан в начале 1970-х годов английской компанией ICL в процессе создания своей операционной системы [234]. Широкую известность эта модель получила после того, как была применена для поддержки разработки систем, написанных на языке Ada. С тех пор многие CASE-средства разрабатываются с использованием общего репозитория.

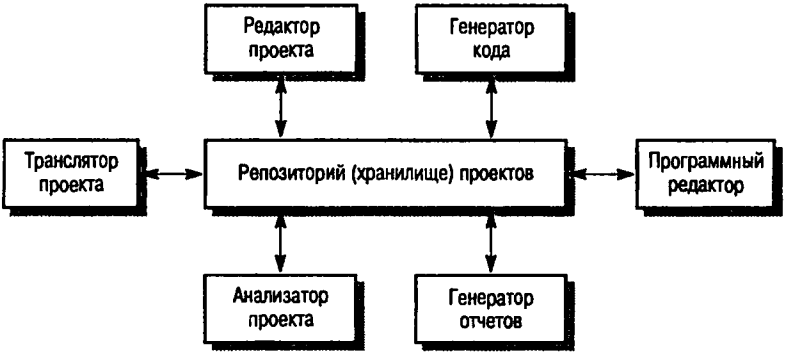


Рис. 10.2. Архитектура интегрированного набора CASE-средств

Совместно используемые репозитории имеют как преимущества, так и недостатки.

1. Очевидно, что совместное использование больших объемов данных эффективно, поскольку не требуется передавать данные из одной подсистемы в другие.
2. С другой стороны, подсистемы должны быть согласованы с моделью репозитория данных. Это всегда приводит к необходимости компромисса между требованиями, предъявляемыми к каждой подсистеме. Компромиссное решение может понизить их производительность. Если форматы данных новых подсистем не подходят под согласованную модель представления данных, интегрировать такие подсистемы сложно или невозможно.
3. Подсистемам, в которых создаются данные, не нужно знать, как эти данные используются в других подсистемах.
4. Поскольку в соответствии с согласованной моделью данных генерируются большие объемы информации, модернизация таких систем проблематична. Перевод системы на новую модель данных будет дорогостоящим и сложным, а порой даже невозможным.
5. В системах с репозиторием такие средства, как резервное копирование, обеспечение безопасности, управление доступом и восстановление данных, централизованы, поскольку входят в систему управления репозиторием. Эти средства выполняют только свои основные операции и не занимаются другими вопросами.
6. С другой стороны, к разным подсистемам предъявляются разные требования, касающиеся безопасности, восстановления и резервирования данных. В модели репозитория ко всем подсистемам применяется одинаковая политика.
7. Модель совместного использования репозитория прозрачна: если новые подсистемы совместимы с согласованной моделью данных, их можно непосредственно интегрировать в систему.
8. Однако сложно разместить репозиторий на нескольких машинах, поскольку могут возникнуть проблемы, связанные с избыточностью и нарушением целостности данных.

В рассматриваемой модели репозиторий является пассивным элементом, а управление им возложено на подсистемы, использующие данные из репозитория. Для систем искусственного интеллекта разработан альтернативный подход. Он основан на модели "рабочей области", которая инициирует подсистемы тогда, когда конкретные данные становятся доступными. Такой подход применим к системам, в которых форма данных хорошо структурирована. Эта модель обсуждается в работе [255].

10.1.2. Модель клиент/сервер

Модель архитектуры клиент/сервер – это модель распределенной системы, в которой показано распределение данных и процессов между несколькими процессорами. Модель включает три основных компонента.

1. Набор автономных серверов, предоставляющих сервисы другим подсистемам. Например, сервер печати, который предоставляет услуги печати, файловые серверы, предоставляющие сервисы управления файлами, и сервер-компилятор, который предлагает сервисы по компилированию исходных кодов программ.
2. Набор клиентов, которые вызывают сервисы, предоставляемые серверами. В контексте системы клиенты являются обычными подсистемами. Допускается параллельное выполнение нескольких экземпляров клиентской программы.
3. Сеть, посредством которой клиенты получают доступ к сервисам. В принципе нет никакого запрета на то, чтобы клиенты и серверы запускались на одной машине. На практике, однако, модель клиент/сервер в такой ситуации не используется.

Клиенты должны знать имена доступных серверов и сервисов, которые они предоставляют. В то же время серверам не нужно знать ни имена клиентов, ни их количество. Клиенты получают доступ к сервисам, предоставляемым сервером, посредством удаленного вызова процедур.

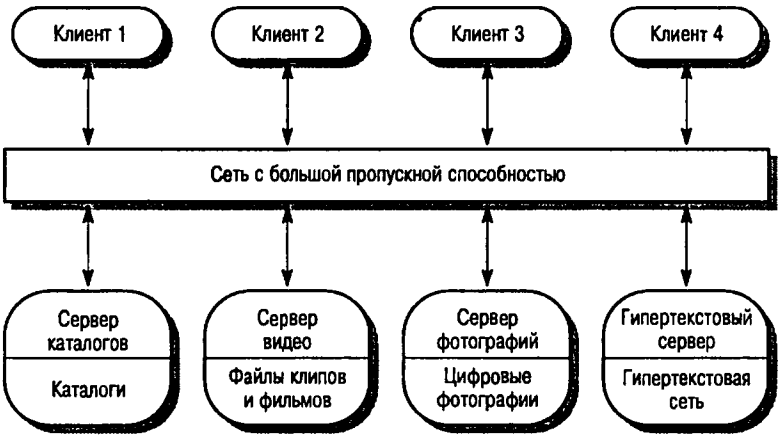


Рис. 10.3. Архитектура библиотечной системы фильмов и фотографий

Пример системы, организованной по типу модели клиент/сервер, показан на рис. 10.3. Это многопользовательская гипертекстовая система, предназначенная для поддержки библиотек фильмов и фотографий. В ней содержится несколько серверов, которые размещают различные типы медиафайлов и управляют ими. Видеофайлы требуется передавать быстро и синхронно, но с относительно малым разрешением. Они могут храниться в сжатом состоянии. Фотографии должны передаваться с высоким разрешением. Каталоги должны обеспечивать работу с множеством запросов и поддерживать связи с использованием гипертекстовой системы. Здесь клиентская программа является просто интегрированным интерфейсом пользователя.

Подход клиент/сервер можно использовать при реализации систем, основанных на репозитории, который поддерживается как сервер системы. Подсистемы, имеющие доступ к репозиторию, являются клиентами. Но обычно каждая подсистема управляет собственными данными. Во время работы серверы и клиенты обмениваются данными, однако при обмене боль-

шими объемами данных могут возникнуть проблемы, связанные с пропускной способностью сети. Правда, с развитием все более быстрых сетей эта проблема теряет свое значение.

Наиболее важное преимущество модели клиент/сервер состоит в том, что она является распределенной архитектурой. Ее эффективно использовать в сетевых системах с множеством распределенных процессоров. В систему легко добавить новый сервер и интегрировать его с остальной частью системы или же обновить серверы, не воздействуя на другие части системы. В главе 11 архитектуры распределенных систем рассматриваются более подробно.

10.1.3. Модель абстрактной машины

Модель архитектуры абстрактной машины (иногда называемая многоуровневой моделью) моделирует взаимодействие подсистем. Она организует систему в виде набора уровней, каждый из которых предоставляет свои сервисы. Каждый уровень определяет *абстрактную машину*, машинный язык которой (сервисы, предоставляемые уровнем) используется для реализации следующего уровня абстрактной машины. Например, наиболее распространенный способ реализации языка программирования состоит в определении идеальной "языковой машины" и компилировании программ, написанных на данном языке, в код этой машины. На следующем шаге трансляции код абстрактной машины конвертируется в реальный машинный код.

Хорошо известным примером такого похода может служить модель OSI[1] сетевых протоколов [352], обсуждаемая в разделе 10.4. Другим примером является трехуровневая модель среды программирования на языке Ada [66]. На рис. 10.4 изображена подобная модель и показано, как с помощью модели абстрактной машины можно представить систему администрирования версий.

Система администрирования версий основана на управлении версиями объектов и предоставляет средства для полного управления конфигурацией системы (см. главу 29). Для поддержки средств управления конфигурацией используется система администрирования объектов, поддерживающая систему базы данных и сервисы управления объектами. В свою очередь, в системе баз данных поддерживаются различные сервисы, например управления транзакциями, отката назад, восстановления и управления доступом. Для управления базами данных используются средства основной операционной системы и ее файловая система.

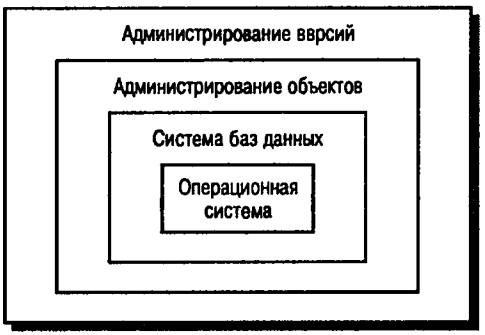


Рис. 10.4. Модель абстрактной машины для системы администрирования версий

[1] *OSI (Open System Interconnection – взаимодействие открытых систем) – международная программа стандартизации обмена данными между компьютерными системами на основе семиуровневой модели протоколов передачи данных в открытых системах. Эта модель предложена Международной организацией по стандартизации ISO (International Standards Organization).* – Прим. ред.

Многоуровневый подход обеспечивает пошаговое развитие систем – при разработке какого-либо уровня предоставляемые им сервисы становятся доступны пользователям. Кроме того, такая архитектура легко изменяема и переносима на разные платформы. Изменение интерфейса любого уровня повлияет только на смежный уровень. Так как в многоуровневых системах зависимости от машинной платформы локализованы на внутренних уровнях, такие системы можно реализовать на других платформах, поскольку потребуется изменить только самые внутренние уровни.

Недостатком многоуровневого подхода является довольно сложная структура системы. Основные средства, такие как управление файлами, необходимые всем абстрактным машинам, предоставляются внутренними уровнями. Поэтому сервисам, запрашиваемым пользователем, возможно, потребуется доступ к внутренним уровням абстрактной машины. Такая ситуация приводит к разрушению модели, так как внешний уровень зависит не только от предшествующего ему уровня, но и от более низких уровней.

10.2. Модели управления

В модели структуры системы показаны все подсистемы, из которых она состоит. Для того чтобы подсистемы функционировали как единое целое, необходимо управлять ими. В структурных моделях нет (и не должно быть) никакой информации по управлению. Однако разработчик архитектуры должен организовать подсистемы согласно некоторой модели управления, которая дополняла бы имеющуюся модель структуры. В моделях управления на уровне архитектуры проектируется поток управления между подсистемами.

Можно выделить два основных типа управления в программных системах.

1. *Централизованное управление.* Одна из подсистем полностью отвечает за управление, запускает и завершает работу остальных подсистем. Управление от первой подсистемы может перейти к другой подсистеме, однако потом обязательно возвращается к первой.
2. *Управление, основанное на событиях.* Здесь вместо одной подсистемы, ответственной за управление, на внешние события может отвечать любая подсистема. События, на которые реагирует система, могут происходить либо в других подсистемах, либо во внешнем окружении системы.

Модель управления дополняет структурные модели. Все описанные ранее структурные модели можно реализовать с помощью централизованного управления или управления, основанного на событиях.

10.2.1. Централизованное управление

В модели централизованного управления одна из систем назначается главной и управляет работой других подсистем. Такие модели можно разбить на два класса, в зависимости от того, последовательно или параллельно реализовано выполнение управляемых подсистем.

1. *Модель вызова-возврата.* Это известная модель организации вызова программных процедур "сверху вниз", в которой управление начинается на вершине иерархии процедур и через вызовы передается на более нижние уровни иерархии. Данная модель применима только в последовательных системах.
2. *Модель диспетчера.* Применяется в параллельных системах. Один системный компонент назначается диспетчером и управляет запуском, завершением и координированием других процессов системы. Процесс (выполняемая подсистема или модуль)

может протекать параллельно с другими процессами. Модель такого типа применима также в последовательных системах, где управляющая программа вызывает отдельные подсистемы в зависимости от значений некоторых переменных состояния. Обычно такое управление реализуется через оператор case.

Модель вызова-возврата представлена на рис. 10.5. Из главной программы можно вызвать подпрограммы 1, 2 и 3, из подпрограммы 1 — подпрограммы 1.1 и 1.2, из подпрограммы 3 — подпрограммы 3.1 и 3.2 и т.д. Такая модель выполнения подпрограмм *не является* структурной — подпрограмма 1.1 не обязательно является частью подпрограммы 1.

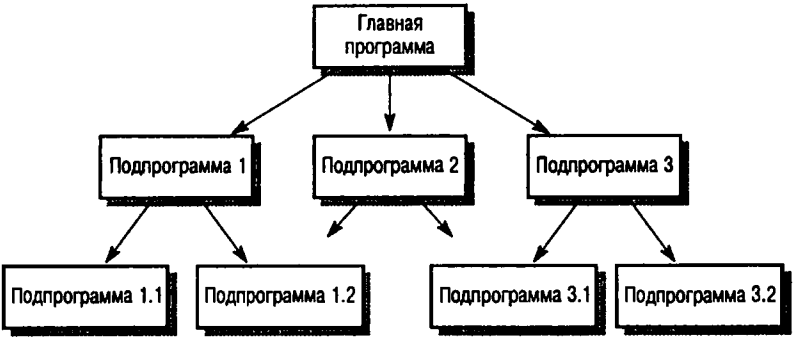


Рис. 10.5. Модель вызова-возврата

Подобная модель встроена в языки программирования Ada, Pascal и C. Управление переходит от программы, расположенной на самом верхнем уровне иерархии, к подпрограмме более нижнего уровня. Затем происходит возврат управления в точку вызова подпрограммы. За управление отвечает та подпрограмма, которая выполняется в текущий момент; она может либо вызывать другие подпрограммы, либо вернуть управление вызвавшей ее подпрограмме. Несовершенство данного стиля программирования при возврате к определенной точке в программе очевидно.

Модель вызова-возврата можно использовать на уровне модулей для управления функциями и объектами. Подпрограммы в языке программирования, которые вызываются из других подпрограмм, являются естественно функциональными. Однако во многих объектно-ориентированных системах операции в объектах (методы) реализованы в виде процедур или функций. Например, объект Java запрашивает сервис из другого объекта посредством вызова соответствующего метода.

Жесткая и ограниченная природа модели вызова-возврата является одновременно и преимуществом и недостатком. Преимущества модели проявляются в относительно простом анализе потоков управления, а также при выборе системы, отвечающей за конкретный ввод данных. Недостаток модели, как вы узнаете из главы 18, состоит в сложной обработке исключительных ситуаций.

На рис. 10.6 представлена модель централизованного управления для параллельной системы. Подобная модель часто используется в "мягких" системах реального времени, в которых нет чересчур строгих временных ограничений. Центральный контроллер управляет выполнением множества процессов, связанных с датчиками и исполнительными механизмами. Система, использующая такую модель управления, рассмотрена в главе 13.

Рис. 10.6. Модель централизованного управления для системы реального времени

Контроллер системы, в зависимости от переменных состояния системы, определяет моменты запуска или завершения процессов. Он проверяет, генерируется ли в остальных процессах информация, для того чтобы затем обработать ее или передать другим процессам на обработку. Обычно контроллер работает постоянно, проверяя датчики и другие процессы или отслеживая изменения состояния, поэтому данную модель иногда называют моделью с обратной связью.

10.2.2. Системы, управляемые событиями

В моделях централизованного управления, как правило, управление системой определяется значениями некоторых переменных ее состояния. В противоположность таким моделям существуют системы, управление которыми основано на внешних событиях. В данном контексте под *событием* подразумевается не только бинарный сигнал типа "да-нет". Здесь сигнал может принимать некоторый диапазон значений. Различие между событием и обычными входными данными заключается в том, что планирование события выходит за рамки управления процессом, обрабатывающим это событие. Для обработки события подсистеме необходим доступ к информации состояния, однако такая информация обычно не определяется потоком управления.

Разработано множество различных типов событийно-управляемых систем. К ним относятся электронные таблицы, в которых изменение значения в какой-либо ячейке изменяет содержимое других ячеек, системы искусственного интеллекта, в которых при выполнении некоторого условия происходит инициирование действия либо используются активные объекты, тогда при изменении значения свойства объекта инициируется некоторое действие. В книге [128] подробно рассматриваются различные типы событийно-управляемых систем.

В этом разделе описаны две модели систем, управляемых событиями.

1. *Модели передачи сообщений.* В этих моделях событие представляет собой передачу сообщения всем подсистемам. Любая подсистема, которая обрабатывает данное событие, отвечает на него.
2. *Модели, управляемые прерываниями.* Такие модели обычно используются в системах реального времени, где внешние прерывания регистрируются обработчиком прерываний, а обрабатываются другим системным компонентом.

Модели передачи сообщений эффективны при интеграции подсистем, распределенных на разных компьютерах, которые объединены в сеть. Модели, управляемые прерываниями, используются в системах реального времени со строгими временными требованиями.

В модели передачи сообщений (рис. 10.7) подсистемы реагируют на определенные события. Если произошло некоторое событие, управление переходит к подсистеме, обрабатывающей данное событие. Между моделью передачи сообщений и моделью централизованного управления, показанной на рис. 10.6, существует отличие: алгоритм управления не встроен в обработчик сообщений и событий. Подсистемы определяют, какие события им требуются, а обработчик сообщений и событий следит, чтобы данные события были отправлены именно им.

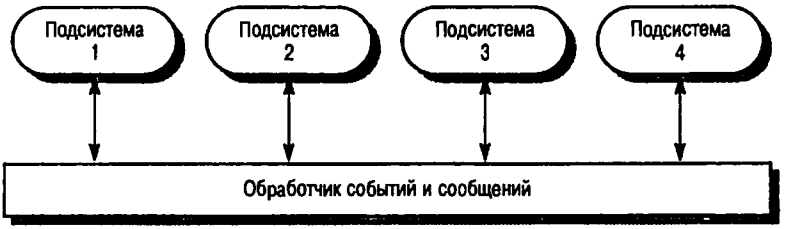


Рис. 10.7. Модель управления, основанная на передаче сообщений

Все события могут генерировать сообщения всем подсистемам, но при этом значительно увеличивается нагрузка при обработке данных. Часто обработчик событий и сообщений управляет регистром подсистем и событиями, на которые они реагируют. Подсистемы генерируют события, в которых, возможно, есть данные для обработки. Обработчик регистрирует событие, принимая во внимание его регистр, и передает это событие в те подсистемы, которые на него реагируют.

Обработчик события всегда поддерживает двухточечное взаимодействие. Поэтому подсистемы могут явно отправить сообщение другой подсистеме. Существует множество разновидностей этой модели [291, 114, 14*]. Брокеры запросов к объектам, обсуждаемые в главе 11, также поддерживают данную модель управления при взаимодействии между распределенными объектами.

Преимуществом модели передачи сообщений является относительно простая модернизация систем, построенных в соответствии с этой моделью. Новую подсистему можно интегрировать в систему, регистрируя ее события в обработчике событий. Каждая подсистема может активизировать любую другую подсистему, не зная ее имени или размещения. Подсистемы также можно реализовать на разных машинах.

Недостатком данной модели является то, что подсистемам неизвестно, когда произойдет обработка события. Генерируя событие, подсистема не знает, какая именно система прореагирует на него. Вполне допустима ситуация, когда разные подсистемы реагируют на одинаковые события. Это может привести к конфликтам при получении доступа к результатам обработки события.

Системы реального времени, в которых одним из требований является быстрая обработка внешних событий, должны быть событийно-управляемыми. Например, система реального времени, управляющая системой безопасности автомобиля, должна определить возможную аварию и успеть наполнить воздухом подушку безопасности до того, как голова водителя ударится о руль. Для того чтобы обеспечить быструю реакцию на события, необходимо использовать управление, основанное на прерываниях.

На рис. 10.8 показана модель управления, основанная на прерываниях. Для каждого типа прерываний существует свой обработчик. Каждый тип прерывания ассоциируется с

ячейкой памяти, в которой хранится адрес обработчика прерывания. При получении определенного прерывания аппаратный переключатель немедленно передает управление обработчику прерывания. В ответ на событие, вызванное прерыванием, обработчик может запустить или завершить другие процессы.

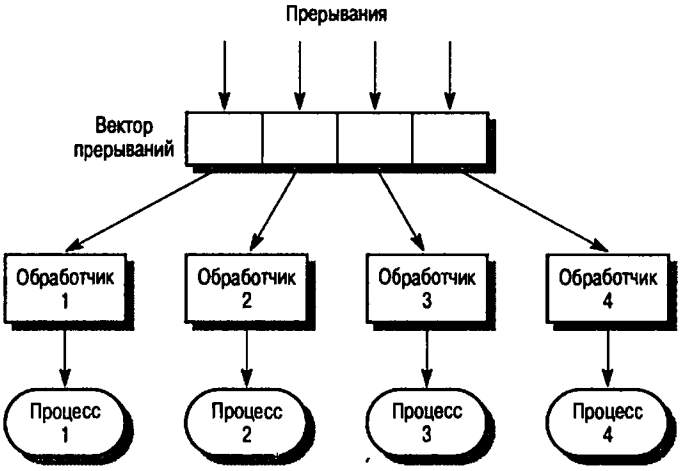


Рис. 10.8. Модель управления, основанная на прерываниях

Данная модель используется только в жестких системах реального времени, где требуется немедленная реакция на определенные события. Можно скомбинировать эту модель с моделью централизованного управления. Центральный диспетчер обрабатывает нормальный ход выполнения системы, а в критических ситуациях используется управление, основанное на прерываниях.

Преимуществом такого подхода является мгновенная реакция системы на происходящие события, недостатками – сложность программирования и аттестации системы. Практически невозможно имитировать все прерывания в процессе тестирования системы. Сложно изменять системы, разработанные на основе такой модели, так как число прерываний ограничено аппаратурой. Никакие другие типы событий не обрабатываются, если достигнут этот предел. Ограничение иногда можно обойти, если на одном прерывании определить несколько типов событий и предоставить для их обработки отдельный обработчик. Однако, если требуется очень быстрая реакция на прерывание, такой подход оказывается непрактичным.

10.3. Модульная декомпозиция

После этапа разработки системной структуры в процессе проектирования следует этап декомпозиции подсистем на модули. Между разбивкой системы на подсистемы и подсистем на модули нет принципиальных отличий. На этом этапе можно использовать модели, рассмотренные в разделе 10.1. Однако компоненты модулей обычно меньше компонентов подсистем, поэтому можно использовать специальные модели декомпозиции.

Здесь рассматриваются две модели, используемые на этапе модульной декомпозиции подсистем.

1. *Объектно-ориентированная модель.* Система состоит из набора взаимодействующих объектов.

2. *Модель потоков данных.* Система состоит из функциональных модулей, которые получают на входе данные и преобразуют их некоторым образом в выходные данные. Такой подход часто называется конвейерным.

В объектно-ориентированной модели модули представляют собой объекты с собственными состояниями и определенными операциями над этими состояниями. В модели потоков данных модули выполняют функциональные преобразования. В обеих моделях модули реализованы либо как последовательные компоненты, либо как процессы.

По возможности разработчикам не стоит принимать поспешных решений о том, будет ли система параллельной или последовательной. Проектирование последовательной системы имеет ряд преимуществ: последовательные программы легче проектировать, реализовать, проверять и тестировать, чем параллельные системы, где очень сложно формализовать, управлять и проверять временные зависимости между процессами. Лучше сначала разбить систему на модули, а на этапе реализации решить, как организовать их выполнение — последовательно или параллельно.

10.3.1. Объектные модели

Объектно-ориентированная архитектурная модель структурирует систему в виде совокупности слабо связанных объектов с четко определенными интерфейсами. Объекты вызывают сервисы, предоставляемые другими объектами. С некоторыми объектными моделями вы познакомились в главе 7, более подробно они рассматриваются в главе 12.

На рис. 10.9 представлен пример объектно-ориентированной архитектурной модели для системы обработки счетов. Данная система выписывает счета заказчикам, получает платежи, отправляет квитанции по поступившим платежам и уведомления по неоплаченным счетам. В этом примере используется система нотации языка моделирования UML (см. главу 7), в которой классы объектов имеют имена и набор атрибутов. Операции, если они есть, определяются в нижней части прямоугольника, обозначающего объект. Штриховые стрелки означают, что объекты используют свойства или сервисы, предоставляемые другими объектами.

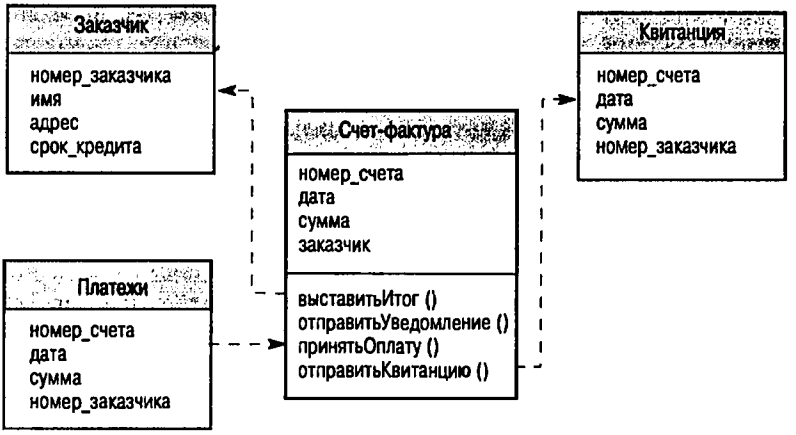


Рис. 10.9. Объектная модель системы обработки счетов

На этапе объектно-ориентированной декомпозиции определяются классы объектов, их свойства и операции. При реализации системы из этих классов создаются объекты; для координации операций объектов используется какая-либо модель управления. В нашем

конкретном примере класс Счет имеет различные связанные операции (методы), которые реализуют функциональные средства системы. Этот класс использует другие классы, представляющие заказчиков, платежи и квитанции.

Преимущества объектно-ориентированного подхода хорошо известны. Поскольку объекты слабо связаны между собой, можно изменять реализацию того или иного объекта, не воздействуя на остальные объекты. Структуру системы легко понять, так как объекты часто являются объектами реального мира. Для непосредственной реализации системных компонентов можно использовать объектно-ориентированные языки программирования.

Вместе с тем объектно-ориентированный подход имеет и недостатки. При использовании сервисов объекты должны явно ссылаться на имена других объектов и знать их интерфейс. Если при изменении системы требуется изменить интерфейс, необходимо оценить эффект от такого изменения с учетом всех пользователей изменяемого объекта. Многие объекты реального мира сложно представить в виде системных объектов.

10.3.2. Модели потоков данных

В управляемой потоками данных модели данные проходят через последовательность преобразований. Каждый шаг обработки данных реализован в виде преобразования. Данные, поступающие на вход системы, проходят через все преобразования и достигают выхода системы. Преобразования могут выполняться последовательно или параллельно. Обработка данных может быть пакетной или поэлементной.

Если преобразования представлены в виде отдельных процессов, такую модель иногда называют конвейером или моделью фильтров, следуя терминологии, принятой в системе Unix. Последняя поддерживает конвейеры, которые действуют как хранилища данных, и набор команд, представляющих функциональные преобразования. Здесь используется термин "фильтр", поскольку преобразование "фильтрует" данные во время обработки потока данных.

Различные варианты модели потоков данных возникли вместе с появлением первых компьютеров, предназначенных для автоматизированной обработки данных. Когда преобразования последовательно обрабатывают пакеты данных, такая архитектурная модель называется пакетной последовательной моделью. Она является основой для многих классов систем обработки данных. Примером могут быть системы (например, системы обработки счетов), которые генерируют большое количество выходных отчетов, полученных с помощью несложных вычислений, но с большим количеством входных записей.

Пример такого типа системной архитектуры показан на рис. 10.10. Здесь организация выписывает счета заказчикам. Раз в неделю платежные квитанции согласуются со счетами. Для оплаченных счетов выдается квитанция. По счетам, не оплаченным в течение установленного срока, выдается соответствующее уведомление.

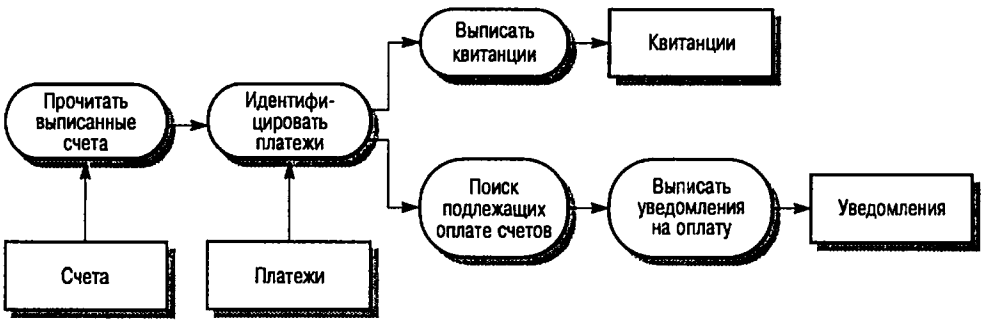


Рис. 10.10. Модель потоков данных для системы обработки счетов

Отметим, что данная модель представляет только часть системы обработки счетов — при выписке счетов используются другие преобразования. Сравните данную модель с объектно-ориентированной, рассмотренной в предыдущем разделе. Объектная модель более абстрактна, так как в ней не содержится информации о последовательности действий.

Данная архитектура имеет ряд преимуществ.

1. Возможность повторного использования преобразований.
2. Понятность, поскольку большинство людей также мыслят в терминах обработки входных и выходных данных.
3. Возможность модификации системы путем непосредственного добавления новых преобразований.
4. Простота реализации как последовательной, так и параллельной систем.

Принципиальный недостаток модели связан с необходимостью использования некоторого общего формата передачи данных, который должен распознаваться всеми преобразованиями. Каждое преобразование либо следует согласовывать со смежными преобразованиями относительно формата обрабатываемых данных, либо нужно предложить стандартный формат для всех обрабатываемых даиных. Каждое преобразование должно выполнять грамматический разбор входных данных и синтезировать выходные данные по соответствующей форме, при этом вычислительная нагрузка на систему возрастает. Невозможно интегрировать преобразования, использующие несовместимые форматы данных.

Взаимодействующие системы трудно описать с помощью модели потоков данных из-за отсутствия прогнозируемого потока обрабатываемых данных. Хотя обычный текстовой ввод и вывод можно смоделировать с помощью модели потоков данных, графические интерфейсы пользователя имеют более сложные форматы ввода-вывода и управлеиие, основанное на разнообразных событиях (например, щелчок кнопкой мыши или выбор из меню). Перевод их в форму, совместимую с моделью потоков данных, является достаточно сложной задачей.

10.4. Проблемно-зависимые архитектуры

Рассмотренные ранее архитектурные модели являются обобщенными. Они широко применяются для многих классов приложений. Наряду с основными моделями, используются архитектурные модели, характерные для конкретной предметной области приложения. Эти модели называются *проблемно-зависимыми архитектурами.*

Можно выделить два типа проблемно-зависимых архитектурных моделей.

1. *Модели классов систем.* Отображают классы реальных систем, вобрав в себя основные характеристики этих классов. Как правило, архитектурные модели классов встречаются в системах реального времени, например в системах сбора данных, мониторинга и т.д.
2. *Базовые модели.* Более абстрактны и предоставляют разработчикам информацию по общей структуре какого-либо типа систем. Например, в работе [298] предложена базовая модель разработки ПО.

Конечно, четких различий между этими видами моделей нет. В некоторых случаях модели классов служат в качестве базовых. Здесь я провожу различие между ними, поскольку базовые модели можно напрямую повторно использовать в проекте. Базовые модели обычно используются в системах коммуникаций и при сравнении возможных системных архитектур. Различны также процессы разработки этих моделей. Модели классов разрабатываются как обобщение существующих систем "снизу вверх", в то время как разработка базовых моделей идет "сверху вниз".

10.4.1. Модели классов систем

Модель компилятора, вероятно, наиболее известный пример архитектурной модели класса систем. В настоящее время разработаны тысячи компиляторов. Считается, что компилятор должен включать перечисленные ниже модули.

1. Лексический анализатор, транслирующий входной язык в некоторый внутренний код.
2. Таблица идентификаторов, выдаваемая лексическим анализатором, в которой содержится информация об используемых в программе именах и типах.
3. Синтаксический анализатор, который проверяет синтаксис компилируемого кода. Он использует заданную грамматику языка и создает синтаксическое дерево.
4. Синтаксическое дерево, которое представляет внутреннюю структуру компилируемой программы.
5. Семантический анализатор, который проверяет семантическую корректность программ на основании информации, полученной из синтаксического дерева и таблицы идентификаторов.
6. Генератор кода, который проходит по синтаксическому дереву и генерирует машинный код.

В компиляторе могут быть и другие компоненты, которые преобразуют синтаксическое дерево, повышают эффективность и устраняют избыточность из сгенерированного машинного кода.

Компоненты, из которых состоит компилятор, можно организовывать в соответствии с разными архитектурными моделями. Можно применить архитектуру потоков данных, в которой таблица идентификаторов служит хранилищем совместно используемых данных. Этот подход отображен на рис. 10.11, здесь этапы лексического, синтаксического и семантического анализа выполняются последовательно.

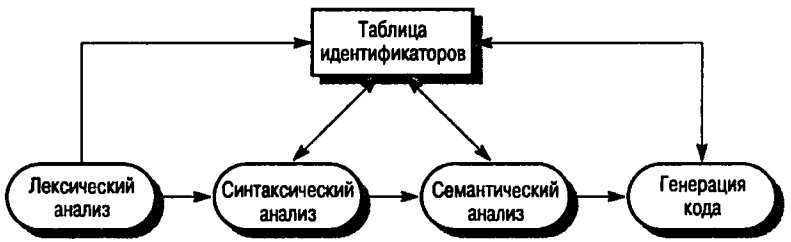


Рис. 10.11. Модель потоков данных для компилятора

Такая модель до сих пор широко применяется. Она эффективна там, где программы компилируются и выполняются без участия пользователя, т.е. в пакетном режиме. Однако такие модели оказываются менее эффективными, если компилятор интегрирован с другими языковыми средствами, например системой редактирования структур, интерактивным отладчиком, программой подготовки печатных документов и т.п. В этом случае, как показано на рис. 10.12, компоненты системы можно организовать в соответствии с моделью репозитория.

В представленной модели компилятора таблица идентификаторов и синтаксическое дерево используются как центральное хранилище информации, или репозиторий, через который взаимодействуют инструментальные средства. Другие данные, такие как грамматические правила и определение выходного формата программы, берутся из инструментальных средств и также помещаются в репозиторий.

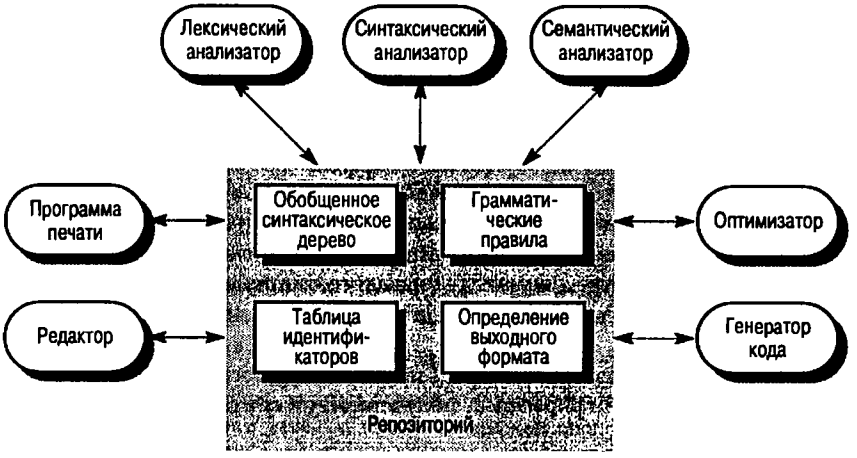


Рис. 10.12. Модель репозитория для компилятора

В настоящее время разработано относительно небольшое количество проблемно-зависимых моделей классов систем. Организации, занимающиеся разработкой подобных моделей, рассматривают их как ценную интеллектуальную собственность, поскольку они являются основой разработки программных систем. Такие модели часто представляют архитектуру целой серии программных продуктов. Например, для всех драйверов принтеров, независимо от свойств конкретного принтера, может использоваться одинаковая исходная архитектура. Подобные архитектуры рассматриваются в главе 14.

10.4.2. Базовые архитектуры

Архитектурные модели классов систем отражают архитектуры уже существующих систем. В противоположность им базовые модели обычно появляются как результат исследований в определенной предметной области приложения. Они представляют собой идеализированную архитектуру, в которой отражены особенности, присущие системам, работающим в данной предметной области.

Примером базовой архитектуры может служить модель OSI [352], являющаяся стандартом взаимодействия открытых систем (краткое описание этой модели приведено в разделе 10.1.3). Если некоторая система совместима с этой моделью, она может взаимодействовать с любыми другими системами, поддерживающими этот стандарт. Таким образом, система управления ассортиментом товаров в супермаркете, разработанная на основе модели OSI, может непосредственно осуществлять обмен данными с системой заказов поставщиков, также разработанной на основе этой модели.

С другой стороны, базовые модели обычно не рассматриваются в качестве методов реализации. Их основное назначение — служить эталоном для сравнения различных систем в какой-либо предметной области, т.е. базовая модель является стандартом при оценке различных систем.

Изображенная на рис. 10.13 модель OSI является семиуровневой моделью взаимодействия открытых систем. Точное назначение разных слоев здесь не существенно. Заметим только, что нижние уровни обеспечивают физическое взаимодействие, средние уровни — передачу данных, а верхние уровни — передачу семантически значимой информации, например стандартизированных документов и т.п.

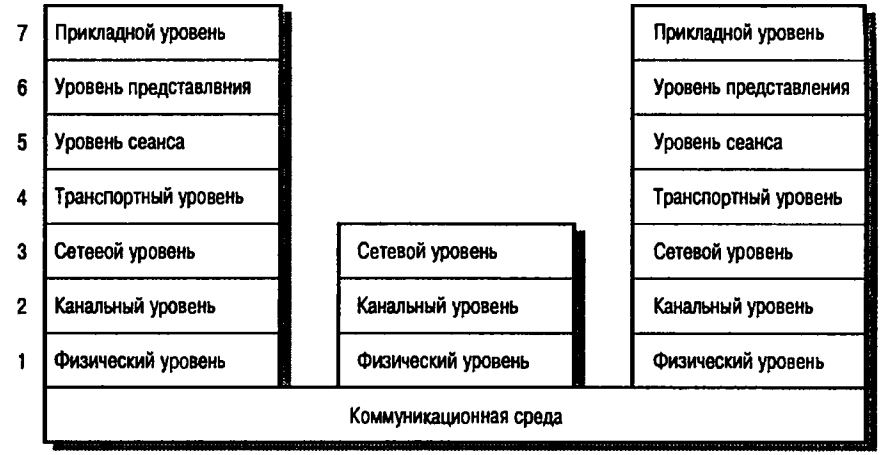


Рис. 10.13. Архитектура базовой модели OSI

Перед разработчиками модели OSI была поставлена конкретная цель — создать стандарт, на основе которого осуществлялось бы взаимодействие между совместимыми системами. Каждый уровень зависел бы только от нижележащего уровня. По мере развития технологии любой из уровней можно было бы реализовать заново, не влияя при этом на другие уровни системы.

На практике многоуровневый подход к архитектурному моделированию носит компромиссный характер. Если различия между компьютерными сетями слишком велики, простое взаимодействие между ними невозможно. Хотя функциональные характеристики каждого уровня четко определены, остаются неопределенными нефункциональные характеристики системы. Поэтому разработчики, реализуя собственные средства высокого уровня, могут "пропустить" некоторые уровни модели. Иначе говоря, чтобы повысить производительность системы, разработчики могут проектировать нестандартные средства, которые не укладываются в рамки модели. После этого простая замена уровней в такой модели вряд ли возможна. Однако от этого эффективность модели не снижается. Данная модель обеспечивает основу для общего структурирования системы и для реализации взаимодействий между системами.

Другими базовыми моделями являются CASE-среды [104, 6*] и модель разработки программного обеспечения [298, 14*]. Некоторые архитектурные шаблоны [124, 13*, 32*] также можно считать базовыми архитектурами. Эти модели обсуждаются в главе 14.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Архитектура ПО — это описание структуры программной системы. Различные архитектурные модели, такие как структурная модель, модель управления и модель модульной декомпозиции, разрабатываются в процессе архитектурного проектирования.
- Большие системы редко сводятся к одной архитектурной модели. Они неоднородны и на разных уровнях обобщения используют разные модели.

- К структурным моделям относятся модели репозитория, модели клиент/сервер и модели абстрактной машины. В модели репозитория совместно используемые данные находятся в общем хранилище (репозитории). В моделях клиент/сервер данные, как правило, распределены. Модели абстрактной машины являются многоуровневыми, причем каждый уровень реализован с помощью средств, предоставляемых предыдущим уровнем.
- К моделям управления относятся модели централизованного управления и модели, управляемые событиями. В централизованных моделях управление зависит от состояния системы; в моделях, управляемых событиями, системой управляют внешние события.
- К моделям модульной декомпозиции относятся модели потоков данных и объектные модели. Модели потоков данных функциональны, в то время как объектные модели являются совокупностью слабо связанных между собой объектов с собственными состояниями и операциями.
- Проблемно-зависимые архитектурные модели отражают свойства и характеристики прикладных областей. Проблемно-зависимые модели делятся на модели классов систем, проектируемые на основе существующих систем, и базовые модели, которые представляют собой идеализированные абстрактные модели для конкретной предметной области.

Упражнения

10.1. Объясните, почему архитектуру системы необходимо разработать до окончания создания спецификации.

10.2. Создайте таблицу, описывающую преимущества и недостатки различных структурных моделей, обсуждавшихся в данной главе.

10.3. Предложите подходящую структурную модель для перечисленных ниже систем. Обоснуйте свой выбор.

- Система автоматической продажи железнодорожных билетов.
- Система видеоконференций, управляемая компьютером, с возможностью одновременного просмотра компьютерных, аудио- и видеоданных несколькими участниками.
- Робот-уборщик, который убирает относительно свободные пространства, например коридоры. Робот должен фиксировать стены и другие преграды.

10.4. На основе выбранной модели разработайте архитектуру для систем из предыдущего упражнения. Сделайте предположения о системных требованиях.

10.5. Объясните, почему модель управления вызова-возврата обычно не подходит для систем реального времени, управляющих определенным процессом.

10.6. Предложите подходящую модель управления для перечисленных ниже систем. Обоснуйте свой выбор.

- Пакетная система обработки данных, которая на основании информации об отработанных часах и ставок заработной платы формирует заявки на зарплату персонала и передает информацию в банк.
- Набор инструментальных программных средств от разных производителей, которые должны работать совместно.
- Телевизионный контроллер, который отвечает на сигналы, поступающие от удаленного блока управления.

10.7. Обсудите преимущества и недостатки модели потоков данных и объектной модели в предположении, что необходимо разработать как локальную, так и распределенную версии программного приложения.

10.8. Существует два набора инструментальных CASE-средств. Необходимо сравнить их. Продумайте, как это сделать с помощью базовой модели CASE-сдеств [61].

10.9. Предположим, существует конкретная должность "архитектор программного обеспечения"; его роль состоит в проектировании системной архитектуры независимо от того, для какого заказчика выполняется данный проект. Такая должность может быть, например, в компании, занимающейся разработкой ПО. Какие трудности могут возникнуть при введении данной должности?

11 Архитектура распределенных систем

Цели

Цель настоящей главы — изучение архитектуры распределенных программных систем. Прочитав эту главу, вы должны:

- знать основные преимущества и недостатки распределенных систем;
- иметь представление о различных подходах, используемых при разработке архитектур клиент/сервер;
- понимать различия между архитектурой клиент/сервер и архитектурой распределенных объектов;
- знать концепцию брокера запросов к объектам и принципы, реализованные в стандартах CORBA.

Содержание

В настоящее время практически все большие программные системы являются распределенными. Распределенной называется такая система, в которой обработка информации сосредоточена не на одной вычислительной машине, а распределена между несколькими компьютерами. При проектировании распределенных систем, которое имеет много общего с проектированием любого другого ПО, все же следует учитывать ряд специфических особенностей. Некоторые из них уже упоминалось во введении к главе 10 при рассмотрении архитектуры клиент/сервер, здесь они обсуждаются более подробно.

Поскольку в наши дни распределенные системы получили широкое распространение, разработчики ПО должны быть знакомы с особенностями их проектирования. До недавнего времени все большие системы в основном являлись централизованными, которые запускались на одной главной вычислительной машине (мэйнфрейме) с подключенными к ней терминалами. Терминалы практически не занимались обработкой информации — все вычисления выполнялись на главной машине. Разработчикам таких систем не приходилось задумываться о проблемах распределенных вычислений.

Все современные программные системы можно разделить на три больших класса.

1. Прикладные программные системы, предназначенные для работы только на одном персональном компьютере или рабочей станции. К ним относятся текстовые процессоры, электронные таблицы, графические системы и т.п.
2. Встроенные системы, предназначенные для работы на одном процессоре либо на интегрированной группе процессоров. К ним относятся системы управления бытовыми устройствами, различными приборами и др.
3. Распределенные системы, в которых программное обеспечение выполняется на слабо интегрированной группе параллельно работающих процессоров, связанных через сеть. К ним относятся системы банкоматов, принадлежащих какому-либо банку, издательские системы, системы ПО коллективного пользования и др.

В настоящее время между перечисленными классами программных систем существуют четкие границы, которые в дальнейшем будут все более стираться. Со временем, когда высокоскоростные беспроводные сети станут широкодоступными, появится возможность динамически интегрировать устройства со встроенными программными системами, например электронные органайзеры с более общими системами.

В книге [81] выделено шесть основных характеристик распределенных систем.

1. *Совместное использование ресурсов.* Распределенные системы допускают совместное использование аппаратных и программных ресурсов, например жестких дисков, принтеров, файлов, компиляторов и т.п., связанных посредством сети. Очевидно, что разделение ресурсов возможно также в многопользовательских системах, однако в этом случае за предоставление ресурсов и их управление должен отвечать центральный компьютер.
2. *Открытость.* Это возможность расширять систему путем добавления новых ресурсов. Распределенные системы — это открытые системы, к которым подключают аппаратное и программное обеспечение от разных производителей.
3. *Параллельность.* В распределенных системах несколько процессов могут одновременно выполняться на разных компьютерах в сети. Эти процессы могут (но не обязательно) взаимодействовать друг с другом во время их выполнения.
4. *Масштабируемость.* В принципе все распределенные системы являются масштабируемыми: чтобы система соответствовала новым требованиям, ее можно наращивать посредством добавления новых вычислительных ресурсов. Но на практике наращивание может ограничиваться сетью, объединяющей отдельные компьютеры системы. Если подключить много новых машин, пропускная способность сети может оказаться недостаточной.

5. *Отказоустойчивость.* Наличие нескольких компьютеров и возможность дублирования информации означает, что распределенные системы устойчивы к определенным аппаратным и программным ошибкам (см. главу 18). Большинство распределенных систем в случае ошибки, как правило, могут поддерживать хотя бы частичную функциональность. Полный сбой в работе системы происходит только в случае сетевых ошибок.
6. *Прозрачность.* Это свойство означает, что пользователям предоставлен полностью прозрачный доступ к ресурсам и в то же время от них скрыта информация о распределении ресурсов в системе. Однако во многих случаях конкретные знания об организации системы помогают пользователю лучше использовать ресурсы.

Разумеется, распределенным системам присущ ряд недостатков.

- *Сложность.* Распределенные системы сложнее централизованных. Намного труднее понять и оценить свойства распределенных систем в целом, а также тестировать эти системы. Например, здесь производительность системы зависит не от скорости работы одного процессора, а от полосы пропускания сети и скорости работы разных процессоров. Перемещая ресурсы из одной части системы в другую, можно радикально повлиять на производительность системы.
- *Безопасность.* Обычно доступ к системе можно получить с нескольких разных машин, сообщения в сети могут просматриваться или перехватываться. Поэтому, в распределенной системе намного сложнее поддерживать безопасность.
- *Управляемость.* Система может состоять из разнотипных компьютеров, на которых могут быть установлены разные версии операционных систем. Ошибки на одной машине могут распространиться на другие машины с непредсказуемыми последствиями. Поэтому требуется значительно больше усилий, чтобы управлять и поддерживать систему в рабочем состоянии.
- *Непредсказуемость.* Как известно всем пользователям Web-сети, реакция распределенных систем на определенные события непредсказуема и зависит от полной загрузки системы, ее организации и сетевой нагрузки. Так как все эти параметры могут постоянно меняться, время, затраченное на выполнение запроса пользователя, в тот или иной момент может существенно различаться.

При обсуждении преимуществ и недостатков распределенных систем в книге [81] определяется ряд критических проблем проектирования таких систем (табл. 11.1). В этой главе основное внимание уделяется архитектуре распределенного ПО, так как я полагаю, что при разработке программных продуктов наиболее значимым является именно этот момент. Если вас интересуют другие темы, обратитесь к специализированным книгам по распределенным системам.

Таблица 11.1. Проблемы проектирования распределенных систем

Проблема проектирования	Описание
Идентификация ресурсов	Ресурсы в распределенной системе располагаются на разных компьютерах, поэтому систему имен ресурсов следует продумать так, чтобы пользователи могли без труда открывать необходимые им ресурсы и ссылаться на них. Примером может служить система унифицированного указателя ресурсов URL, которая определяет адреса Web-страниц. Без легковоспринимаемой и универсальной системы идентификации большая часть ресурсов окажется недоступной пользователям системы

Окончание табл. 11.1

Проблема проектирования	Описание
Коммуникации	Универсальная работоспособность Internet и эффективная реализация протоколов TCP/IP в Internet для большинства распределенных систем служат примером наиболее эффективного способа организации взаимодействия между компьютерами. Однако там, где на производительность, надежность и прочее накладываются специальные требования, можно воспользоваться альтернативными способами системных коммуникаций
Качество системиого сервиса	Качество сервиса, предлагаемое системой, отражает ее производительность, работоспособность и надежность. На качество сервиса влияет целый ряд факторов: распределение системных процессов, распределение ресурсов, системные и сетевые аппаратные средства и возможности адаптации системы
Архитектура программного обеспечения	Архитектура программного обеспечения описывает распределение системных функций по компонентам системы, а также распределение этих компонентов по процессорам. Если необходимо поддерживать высокое качество системного сервиса, выбор правильной архитектуры оказывается решающим фактором

Задача разработчиков распределенных систем — спроектировать программное или аппаратное обеспечение так, чтобы предоставить все необходимые характеристики распределенной системы. А для этого требуется знать преимущества и недостатки различных архитектур распределенных систем. Здесь выделяется два родственных типа архитектур распределенных систем.

1. *Архитектура клиент/сервер.* В этой модели систему можно представить как набор сервисов, предоставляемых серверами клиентам. В таких системах серверы и клиенты значительно отличаются друг от друга.
2. *Архитектура распределенных объектов.* В этом случае между серверами и клиентами нет различий и систему можно представить как набор взаимодействующих объектов, местоположение которых не имеет особого значения. Между поставщиком сервисов и их пользователями не существует различий.

В распределенной системе разные системные компоненты могут быть реализованы на разных языках программирования и выполняться на разных типах процессоров. Модели данных, представление информации и протоколы взаимодействия — все это не обязательно будет однотипным в распределенной системе. Следовательно, для распределенных систем необходимо такое программное обеспечение, которое могло бы управлять этими разнотипными частями и гарантировать взаимодействие и обмен данными между ними. *Промежуточное программное обеспечение* относится именно к такому классу ПО. Оно находится как бы посередине между разными частями распределенных компонентов системы.

В статье [37] описаны различные типы промежуточного ПО, которое может поддерживать распределенные вычисления. Как правило, такое ПО составляется из готовых компонентов и не требует от разработчиков специальных доработок. В качестве примеров промежуточного ПО можно привести программы управления взаимодействием с базами данных, менеджеры транзакций, преобразователи данных, коммуникационные инспекторы и др. Далее в главе будет описаиа структура распределенных систем как класс промежуточного ПО.

Распределенные системы обычно разрабатываются на основе объектно-ориентированного подхода. Эти системы создаются из слабо интегрированных частей, каждая из которых может непосредственно взаимодействовать как с пользователем, так и с другими частями системы. Эти части по возможности должны реагировать на независимые события. Программные объекты, построенные на основе таких принципов, являются естественными компонентами распределенных систем. Если вы еще не знакомы с концепцией объектов, рекомендую сначала прочитать главу 12, а затем вновь вернуться к данной главе.

11.1. Многопроцессорная архитектура

Самой простой распределенной системой является многопроцессорная система. Она состоит из множества различных процессов, которые могут (но не обязательно) выполняться на разных процессорах. Данная модель часто используется в больших системах реального времени. Как вы узнаете из главы 13, эти системы собирают информацию, принимают на ее основе решения и отправляют сигналы исполнительному механизму, который изменяет системное окружение. В принципе все процессы, связанные со сбором информации, принятием решений и управлением исполнительным механизмом, могут выполняться на одном процессоре под управлением планировщика заданий. Использование нескольких процессоров повышает производительность системы и ее способность к восстановлению. Распределение процессов между процессорами может переопределяться (присуще критическим системам) или же находиться под управлением диспетчера процессов.

На рис. 11.1 показан пример системы такого типа. Это упрощенная модель системы управления транспортным потоком. Группа распределенных датчиков собирает информацию о величине потока. Собранные данные перед отправкой в диспетчерскую обрабатываются на месте. На основании полученной информации операторы принимают решения и управляют светофорами. В этом примере для управления датчиками, диспетчерской и светофорами имеются отдельные логические процессы. Это могут быть как отдельные процессы, так и группа процессов. В нашем примере они выполняются на разных процессорах.

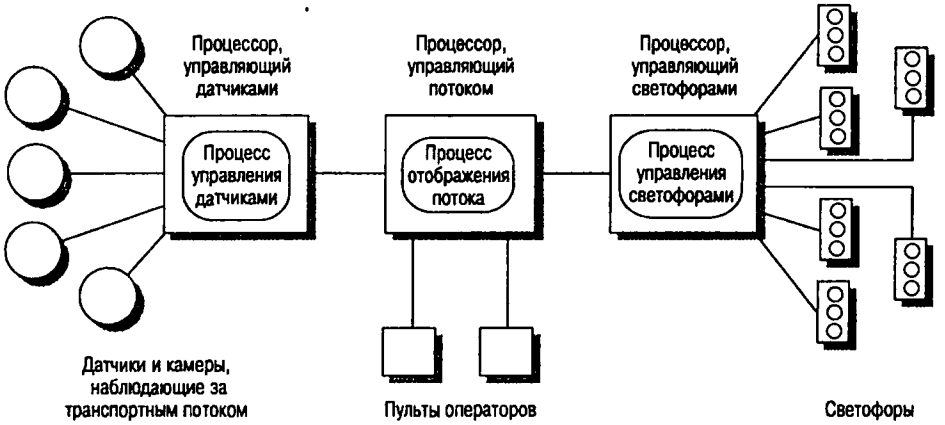


Рис. 11.1. Многопроцессорная система управления движением транспорта

Системы ПО, одновременно выполняющие множество процессов, не обязательно являются распределенными. Если в системе более одного процессора, реализовать распределение процессов не представляет труда. Однако при создании многопроцессорных программных систем не обязательно отталкиваться только от распределенных систем. При проектировании систем такого типа, по существу, используется тот же подход, что и при проектировании систем реального времени, которые рассматриваются в главе 13.

11.2. Архитектура клиент/сервер

В главе 10 уже рассматривалась концепция клиент/сервер. В архитектуре клиент/сервер программное приложение моделируется как набор сервисов, предоставляемых серверами, и множество клиентов, использующих эти сервисы [264, 2*]. Клиенты должны знать о доступных (имеющихся) серверах, хотя могут и не иметь представления о существовании других клиентов. Как видно из рис. 11.2, на котором представлена схема распределенной архитектуры клиент/сервер, клиенты и серверы представляют разные процессы.

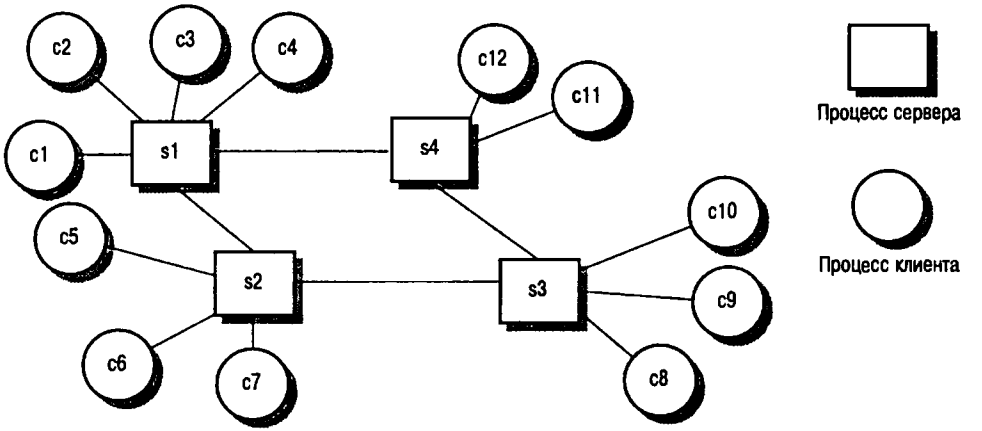


Рис. 11.2. Система клиент/сервер

В системе между процессами и процессорами не обязательно должно соблюдаться отношение "один к одному". На рис. 11.3 показана физическая архитектура системы, которая состоит из шести клиентских машин и двух серверов. На них запускаются клиентские и серверные процессы, изображенные на рис. 11.2. В общем случае, говоря о клиентах и серверах, я подразумеваю скорее логические процессы, чем физические машины, на которых выполняются эти процессы.

Архитектура системы клиент/сервер должна отражать логическую структуру разрабатываемого программного приложения. На рис. 11.4 предлагается еще один взгляд на программное приложение, структурированное в виде трех уровней. Уровень представления обеспечивает информацию для пользователей и взаимодействие с ними. Уровень выполнения приложения реализует логику работы приложения. На уровне управления данными выполняются все операции с базами данных. В централизованных системах между этими уровнями нет четкого разделения. Однако при проектировании распределенных систем необходимо разделять эти уровни, чтобы затем расположить каждый уровень на разных компьютерах.

Самой простой архитектурой клиент/сервер является двухуровневая, в которой приложение состоит из сервера (или множества идентичных серверов) и группы клиентов. Существует два вида такой архитектуры (рис. 11.5).

1. *Модель тонкого клиента.* В этой модели вся работа приложения и управление данными выполняются на сервере. На клиентской машине запускается только ПО уровня представления.

2. *Модель толстого клиента.* В этой модели сервер только управляет данными. На клиентской машине реализована работа приложения и взаимодействие с пользователем системы.

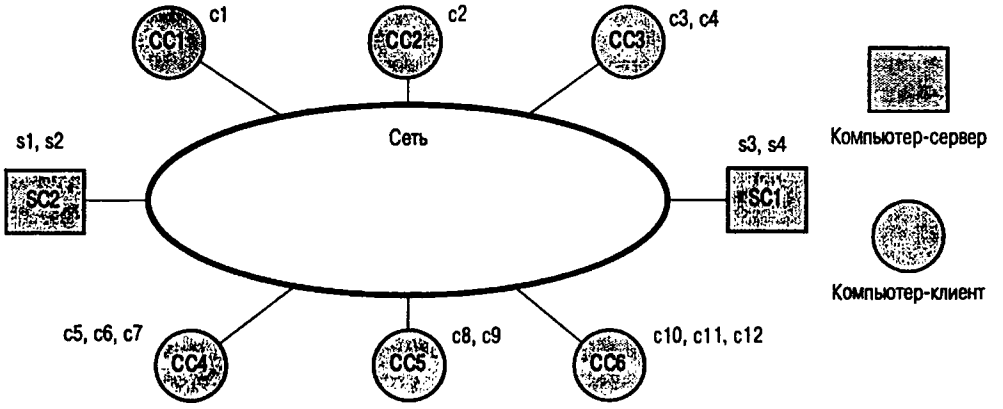


Рис. 11.3. Компьютеры в сети клиент/сервер

Рис. 11.4. Уровни программного приложения

Тонкий клиент двухуровневой архитектуры – самый простой способ перевода существующих централизованных систем (см. главу 26) в архитектуру клиент/сервер. Пользовательский интерфейс в этих системах "переселяется" на персональный компьютер, а само программное приложение выполняет функции сервера, т.е. выполняет все процессы приложения и управляет данными. Модель тонкого клиента можно также реализовать там, где клиенты представляют собой обычные сетевые устройства, а не персональные компьютеры или рабочие станции. Сетевые устройства запускают Internet-броузер и пользовательский интерфейс, реализованный внутри системы.

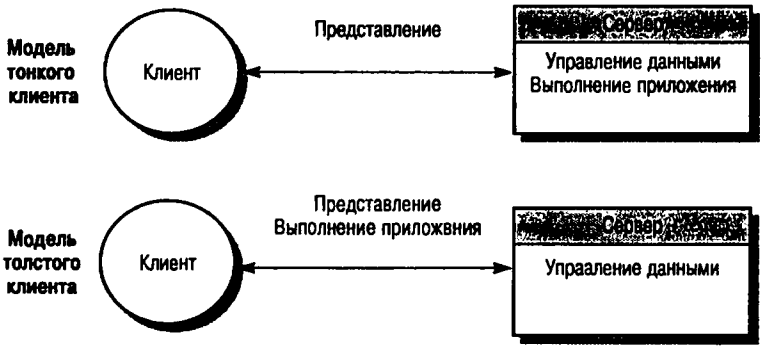


Рис. 11.5. Модели тонкого и толстого клиентов

Главный недостаток модели тонкого клиента – большая загруженность сервера и сети. Все вычисления выполняются на сервере, а это может привести к значительному сетевому трафику между клиентом и сервером. В современных компьютерах достаточно вычислительной мощности, но она практически не используется в модели тонкого клиента банка.

Напротив, модель толстого клиента использует вычислительную мощность локальных машин: и уровень выполнения приложения, и уровень представления помещаются на клиентский компьютер. Сервер здесь, по существу, является сервером транзакций, который управляет всеми транзакциями баз данных. Примером архитектуры такого типа могут служить системы банкоматов, в которых банкомат является клиентом, а сервер — центральным компьютером, обслуживающим базу данных по расчетам с клиентами.

На рис. 11.6 показана сетевая система банкоматов. Заметим, что банкоматы связаны с базой данных расчетов не напрямую, а через монитор телеобработки. Этот монитор является промежуточным звеном, которое взаимодействует с удаленными клиентами и организует запросы клиентов в последовательность транзакций для работы с базой данных. Использование последовательных транзакций при возникновении сбоев позволяет системе восстановиться без потери данных.

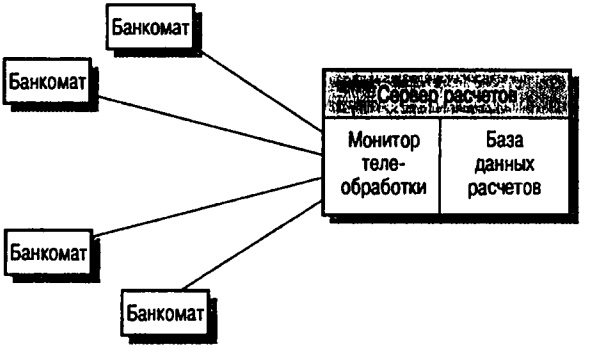


Рис. 11.6. Система клиент/сервер для сети банкоматов

Поскольку в модели толстого клиента выполнение программного приложения организовано более эффективно, чем в модели тонкого клиента, управлять такой системой сложнее. Здесь функции приложения распределены между множеством разных машин. Необходимость замены приложения приводит к его повторной инсталляции на всех клиентских компьютерах, что требует больших расходов, если в системе сотни клиентов.

Появление языка Java и загружаемых аплетов позволили разрабатывать модели клиент/сервер, которые находятся где-то посередине между моделями тонкого и толстого клиента. Часть программ, составляющих приложение, можно загружать на клиентской машине как аплеты Java и тем самым разгрузить сервер. Интерфейс пользователя строится посредством Web-броузера, который запускает аплеты Java. Однако Web-броузеры от различных производителей и даже различные версии Web-броузеров от одного производителя не всегда выполняются одинаково. Более ранние версии броузеров на старых машинах не всегда могут запустить аплеты Java. Следовательно, такой подход можно использовать только тогда, когда вы уверены, что у всех пользователей системы установлены броузеры, совместимые с Java.

В двухуровневой модели клиент/сервер существенной проблемой является размещение на двух компьютерных системах трех логических уровней – представления, выполнения приложения и управления данными. Поэтому в данной модели часто возникают либо проблемы с масштабируемостью и производительностью, если выбрана модель тонкого клиента, либо проблемы, связанные с управлением системой, если используется модель толстого клиента. Чтобы избежать этих проблем, необходимо применить альтернативный подход – трехуровневую модель архитектуры клиент/сервер (рис. 11.7). В этой архитектуре уровням представления, выполнения приложения и управления данными соответствуют отдельные процессы.

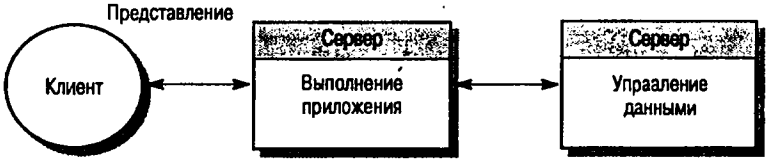


Рис. 11.7. Трехуровневая архитектура клиент/сервер

Архитектура ПО, построенная по трехуровневой модели клиент/сервер, не требует, чтобы в сеть были объединены три компьютерных системы. На одном компьютере-сервере можно запустить и выполнение приложения, и управление данными как отдельные логические серверы. В то же время, если требования к системе возрастут, можно будет относительно просто разделить выполнение приложения и управление данными и выполнять их на разных процессорах.

Банковскую систему, использующую Internet-сервисы, можно реализовать с помощью трехуровневой архитектуры клиент/сервер. База данных расчетов (обычно расположенная на главном компьютере) предоставляет сервисы управления данными, Web-сервер поддерживает сервисы приложения, например средства перевода денег, генерацию отчетов, оплату счетов и др. А компьютер пользователя с Internet-броузером является клиентом. Как показано на рис. 11.8, эта система масштабируема, так как в нее относительно просто добавить новые Web-серверы при увеличении количества клиентов.

Использование трехуровневой архитектуры в этом примере позволило оптимизировать передачу данных между Web-сервером и сервером базы данных. Взаимодействие между этими системами не обязательно строить на стандартах Internet, можно использовать более быстрые коммуникационные протоколы низкого уровня. Обычно информацию от базы данных обрабатывает эффективное промежуточное ПО, которое поддерживает запросы к базе данных на языке структурированных запросов SQL.

В некоторых случаях трехуровневую модель клиент/сервер можно перевести в многоуровневую, добавив в систему дополнительные серверы. Многоуровневые системы можно использовать и там, где приложениям необходимо иметь доступ к информации, находящейся в разных базах данных. В этом случае объединяющий сервер располагается между

сервером, на котором выполняется приложение, и серверами баз данных. Объединяющий сервер собирает распределенные данные и представляет их в приложении таким образом, будто они находятся в одной базе данных.

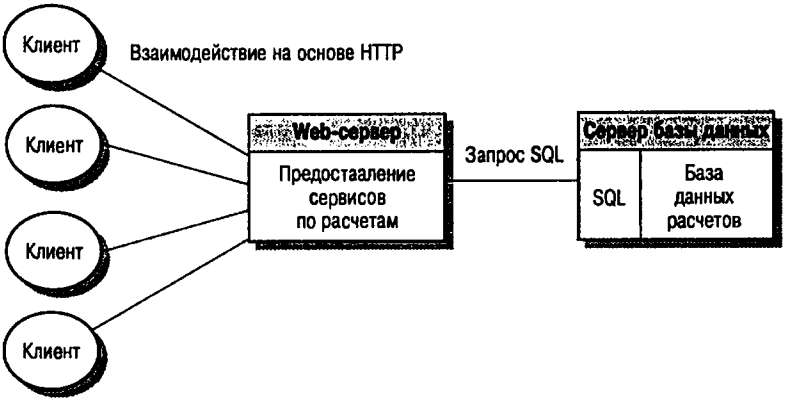


Рис. 11.8. Распределенная архитектура банковской системы с использованием Internet-сервисов

Разработчики архитектур клиент/сервер, выбирая наиболее подходящую, должны учитывать ряд факторов. В табл. 11.2 перечислены различные случаи применения архитектуры клиент/сервер.

Таблица 11.2. Применение разных типов архитектуры клиент/сервер

Архитектура	Приложения
Двухуровневая архитектура тонкого клиента	Наследуемые системы, в которых нецелесообразно разделять выполнение приложения и управления данными.
	Приложения с интенсивными вычислениями, например компиляторы, но с незначительным объемом управления данными.
	Приложения, в которых обрабатываются большие массивы данных (запросы), но с небольшим объемом вычислений в самом приложении
Двухуровневая архитектура толстого клиента	Приложения, где пользователю требуется интенсивная обработка данных (например, визуализация данных или большие объемы вычисления).
	Приложения с относительно постоянным набором функций на стороне пользователя, применяемых в среде с хорошо отлаженным системным управлением
Трехуровневая и многоуровневая архитектуры клиент/сервер	Большие приложения с сотнями и тысячами клиентов.
	Приложения, в которых часто меняются и данные, и методы обработки.
	Приложения, в которых выполняется интеграция данных из многих источников

11.3. Архитектура распределенных объектов

В модели клиент/сервер распределенной системы между клиентами и серверами существуют различия. Клиент запрашивает сервисы только у сервера, но не у других клиентов; серверы могут функционировать как клиенты и запрашивать сервисы у других серверов, но не у клиентов; клиенты должны знать о сервисах, предоставляемых определенными серверами, и о том, как взаимодействуют эти серверы. Такая модель отлично подходит ко многим типам приложений, но в то же время ограничивает разработчиков системы, которые вынуждены решать, где предоставлять сервисы. Они также должны обеспечить поддержку масштабируемости и разработать средства включения клиентов в систему на распределенных серверах.

Более общим подходом, применяемым в проектировании распределенных систем, является стирание различий между клиентом и сервером и проектирование архитектуры системы как архитектуры распределенных объектов. В этой архитектуре (рис. 11.9) основными компонентами системы являются объекты, предоставляющие набор сервисов через свои интерфейсы. Другие объекты вызывают эти сервисы, не делая различий между клиентом (пользователем сервиса) и сервером (поставщиком сервиса).

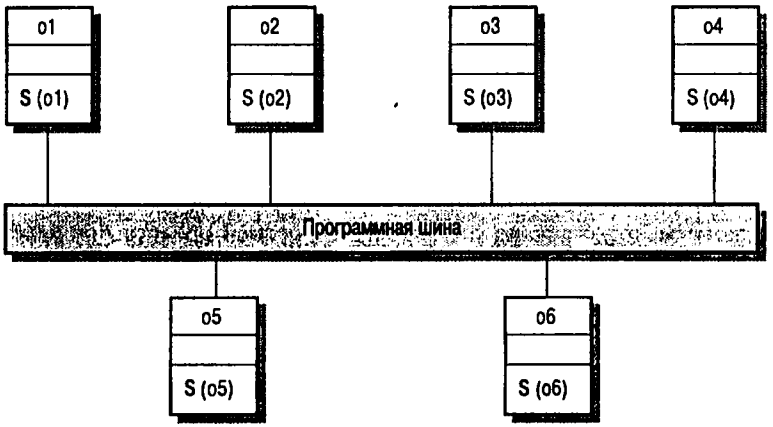


Рис. 11.9. Архитектура распределенных объектов

Объекты могут располагаться на разных компьютерах в сети и взаимодействовать посредством промежуточного ПО. По аналогии с системной шиной, которая позволяет подключать различные устройства и поддерживать взаимодействие между аппаратными средствами, промежуточное ПО можно рассматривать как шину программного обеспечения. Она предоставляет набор сервисов, позволяющий объектам взаимодействовать друг с другом, добавлять или удалять их из системы. Промежуточное ПО называют брокером запросов к объектам. Его задача — обеспечивать интерфейс между объектами. Брокеры запросов к объектам рассматриваются в разделе 11.4.

Ниже перечислены основные преимущества модели архитектуры распределенных объектов.

- Разработчики системы могут не спешить с принятием решений относительно того, где и как будут предоставляться сервисы. Объекты, предоставляющие сервисы, могут выполняться в любом месте (узле) сети. Следовательно, различие между моделями

толстого и тонкого клиентов становятся несущественными, так как нет необходимости заранее планировать размещение объектов для выполнения приложения.

- Системная архитектура достаточно открыта, что позволяет при необходимости добавлять в систему новые ресурсы. В следующем разделе отмечается, что стандарты программной шины постоянно совершенствуются, что позволяет объектам, написанным на разных языках программирования, взаимодействовать и предоставлять сервисы друг другу.
- Гибкость и масштабируемость системы. Для того чтобы справиться с системными нагрузками, можно создавать экземпляры системы с одинаковыми сервисами, которые будут предоставляться разными объектами или разными экземплярами (копиями) объектов. При увеличении нагрузки в систему можно добавить новые объекты, не прерывая при этом работу других ее объектов.
- Существует возможность динамически переконфигурировать систему посредством объектов, мигрирующих в сети по запросам. Объекты, предоставляющие сервисы, могут мигрировать на тот же процессор, что и объекты, запрашивающие сервисы, тем самым повышая производительность системы.

В процессе проектирования систем архитектуру распределенных объектов можно использовать двояко.

1. В виде логической модели, которая позволяет разработчикам структурировать и спланировать систему. В этом случае функциональность приложения описывается только в терминах и комбинациях сервисов. Затем разрабатываются способы предоставления сервисов с помощью нескольких распределенных объектов. На этом уровне, как правило, проектируют крупномодульные объекты, которые предоставляют сервисы, отражающие специфику конкретной области приложения. Например, в программу учета розничной торговли можно включить объекты, которые бы вели учет состояния запасов, отслеживали взаимодействие с клиентами, классифицировали товары и др.
2. Как гибкий подход к реализации систем клиент/сервер. В этом случае логическая модель системы — это модель клиент/сервер, в которой клиенты и серверы реализованы как распределенные объекты, взаимодействующие посредством программной шины. При таком подходе легко заменить систему, например двухуровневую на многоуровневую. В этом случае ни сервер, ни клиент не могут быть реализованы в одном объекте, однако могут состоять из множества небольших объектов, каждый из которых предоставляет определенный сервис.

Примером системы, которой подходит архитектура распределенных объектов, может служить система обработки данных, хранящихся в разных базах данных (рис. 11.10). В этом примере любую базу данных можно представить как объект с интерфейсом, предоставляющим доступ к данным "только чтение". Каждый из объектов-интеграторов занимается определенными типами зависимостей между данными, собирая информацию из баз данных, чтобы попытаться проследить эти зависимости.

Объекты-визуализаторы взаимодействуют с объектами-интеграторами для представления данных в графическом виде либо для составления отчетов по анализируемым данным. Способы представление графической информации рассматриваются в главе 15.

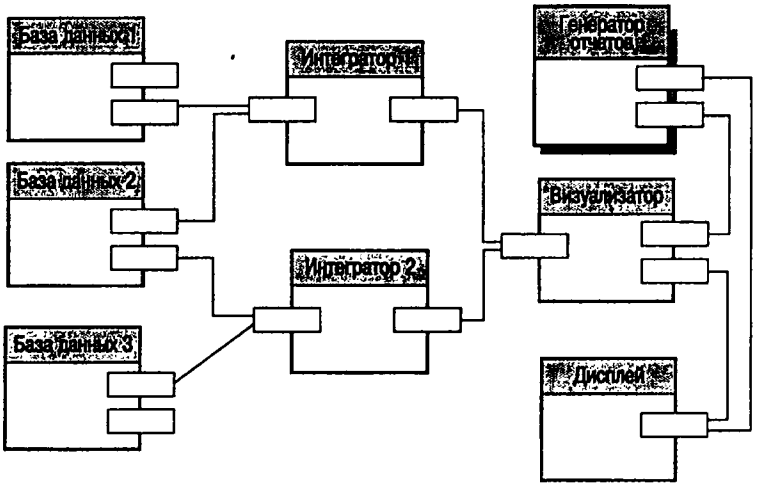


Рис. 11.10. Архитектура распределенной системы обработки данных

Для такого типа приложений архитектура распределенных объектов подходит больше, чем архитектура клиент/сервер, по трем причинам.

1. В этих системах (в отличие, например, от системы банкоматов) нет одного поставщика сервиса, на котором были бы сосредоточены все сервисы управления данными.
2. Можно увеличивать количество доступных баз данных, не прерывая работу системы, поскольку каждая база данных представляет собой просто объект. Эти объекты поддерживают упрощенный интерфейс, который управляет доступом к данным. Доступные базы данных можно разместить на разных машинах.
3. Посредством добавления новых объектов-интеграторов можно отслеживать новые типы зависимостей между данными.

Главным недостатком архитектур распределенных объектов является то, что их сложнее проектировать, чем системы клиент/сервер. Оказывается, что системы клиент/сервер предоставляют более естественный подход к созданию распределенных систем. В нем отражаются взаимоотношения между людьми, при которых одни люди пользуются услугами других людей, специализирующихся на предоставлении конкретных услуг. Намного труднее разработать систему в соответствии с архитектурой распределенных объектов, поскольку индустрия создания ПО пока еще не накопила достаточного опыта в проектировании и разработке крупномодульных объектов.

11.4. CORBA

Как уже отмечалось в предыдущем разделе, при реализации архитектуры распределенных объектов необходимо промежуточное программное обеспечение (брокеры запросов к объектам), организующее взаимодействие между распределенными объектами. Здесь могут возникнуть определенные проблемы, поскольку объекты в системе могут быть реализованы на разных языках программирования, могут запускаться на разных платформах и их имена не должны быть известны всем другим объектам системы. Поэтому промежуточное ПО должно выполнять большую работу для того, чтобы поддерживалось постоянное взаимодействие объектов.

В настоящий момент для поддержки распределенных объектных вычислений существует два основных стандарта промежуточного ПО.

1. CORBA (Common Object Request Broker Architecture – архитектура брокеров запросов к общим объектам). Это набор стандартов для промежуточного ПО, разработанный группой OMG (Object Management Group – группа по управлению объектами). OMG является консорциумом фирм-производителей программного и аппаратного обеспечения, в числе которых такие компании, как Sun, Hewlett-Packard и IBM. Стандарты CORBA определяют общий машинонезависимый подход к распределенным объектным вычислениям. Разными производителями разработано множество реализаций этого стандарта. Стандарты CORBA поддерживаются операционной системой Unix и операционными системами от Microsoft.
2. DCOM (Distributed Component Object Model – объектная модель распределенных компонентов). DCOM представляет собой стандарт, разработанный и реализованный компанией Microsoft и интегрированный в ее операционные системы. Данная модель распределенных вычислений менее универсальна, чем CORBA и предлагает более ограниченные возможности сетевых взаимодействий. В настоящий момент использование DCOM ограничивается операционными системами Microsoft.

Здесь я решил уделить внимание технологии CORBA, поскольку она более универсальна. Кроме того, я считаю, что, вероятно, CORBA, DCOM и другие технологии, например RMI (Remote Method Invocation – вызов удаленного метода, технология построения распределенных приложений на языке Java), будут постепенно сближаться друг с другом и это сближение будет базироваться на стандартах CORBA. Поэтому нет необходимости в еще одном стандарте. Различные стандарты будут только помехой в дальнейшем развитии.

Стандарты CORBA определены группой OMG, которая объединяет более 500 компаний, поддерживающих объектно-ориентированные разработки. Роль OMG – создание стандартов для объектно-ориентированных разработок, а не обеспечение конкретных реализаций этих стандартов. Эти стандарты находятся в свободном доступе на Web-узле OMG. Группа занимается не только стандартами CORBA, но также определяет широкий диапазон других стандартов, включая язык моделирования UML.

Представление распределенных приложений в рамках CORBA показано на рис. 11.11. Это упрощенная схема архитектуры управления объектами, взятая из статьи [317]. Предполагается, что распределенное приложение должно состоять из перечисленных ниже компонентов.

1. Объекты приложения, которые созданы и разработаны для данного программного продукта.
2. Стандартные объекты, которые определены группой OMG для специфических задач. Во время написания книги множество специалистов занимались разработкой стандартов объектов в области финансирования, страхования, электронной коммерции, здравоохранения и многих других.
3. Основные сервисы CORBA, поддерживающие базовые сервисы распределенных вычислений, например каталоги, управление защитой и др.
4. Горизонтальные средства CORBA, например пользовательские интерфейсы, средства управления системой и т.п. Под горизонтальными подразумеваются средства, общие для многих приложений.

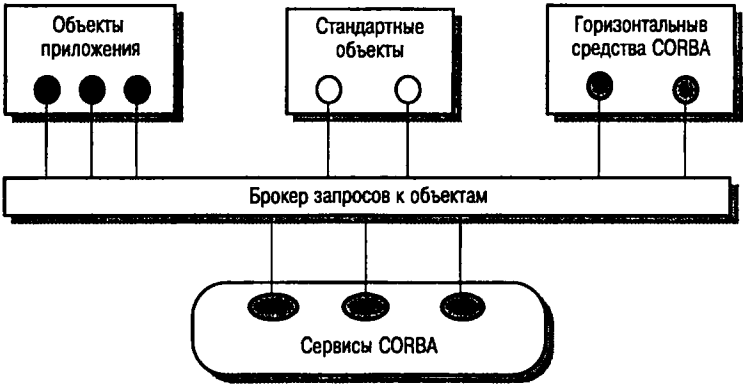


Рис. 11.11. Структура распределенного приложения, основанного на стандартах CORBA

Стандарты CORBA описывают четыре основных элемента.

1. Модель объектов, в которой объект CORBA инкапсулирует состояния посредством четкого описания на языке IDL (Interface Definition Language – язык описания интерфейсов).
2. Брокер запросов к объектам (Object Request Broker – ORB), который управляет запросами к сервисам объектов. ORB размещает объекты, предоставляющие сервисы, подготавливает их к получению запросов, передает запрос к сервису и возвращает результаты объекту, сделавшему запрос.
3. Совокупность сервисов объектов, которые являются основными сервисами, и необходимы во многих распределенных приложениях. Примерами могут быть службы каталогов, сервисы транзакций и сервисы поддержки временных объектов.
4. Совокупность общих компонентов, построенных на верхнем уровне основных сервисов. Они могут быть как вертикальными, отражающими специфику конкретной области, так и горизонтальными универсальными компонентами, используемыми во многих программных приложениях. Эти компоненты рассматриваются в главе 14.

В модели CORBA объект инкапсулирует атрибуты и сервисы как обычный объект. Вместе с тем в объектах CORBA еще должно содержаться определение различных интерфейсов, описывающих глобальные атрибуты и операции объекта. Интерфейсы объектов CORBA определяются на стандартном универсальном языке описания интерфейсов IDL. Если один объект запрашивает сервисы, предоставляемые другими объектами, он получает доступ к этим сервисам через IDL-интерфейс. Объекты CORBA имеют уникальный идентификатор, называемый IOR (Interoperable Object Reference – ссылка на взаимодействующий объект). Когда один объект отправляет запросы к сервису, предоставляемому другим объектом, используется идентификатор IOR.

Брокеру запросов к объектам известны объекты, запрашивающие сервисы и их интерфейсы. Он организует взаимодействие между объектами. Взаимодействующим объектам не требуется что-либо знать о размещении других объектов, а также об их реализации. Так как интерфейс IDL отделяет объекты от брокера, реализацию объектов можно изменять, не затрагивая другие компоненты системы.

На рис. 11.12 показано, как объекты o1 и o2 взаимодействуют посредством брокера запросов к объектам. Вызывающий объект (o1) связан с заглушкой (stub) IDL, которая опре-

деляет интерфейс объекта, предоставляющего сервис. Конструктор объекта o1 при запросе к сервису внедряет вызовы в заглушку своей реализации объекта. Язык IDL является расширением C++, поэтому, если вы программируете на языках C++, C или Java, получить доступ к заглушке совсем просто. Перевод описания интерфейса объекта на IDL также возможен и для других языков, например Ada или COBOL. Но в этих случаях необходима соответствующая инструментальная поддержка.

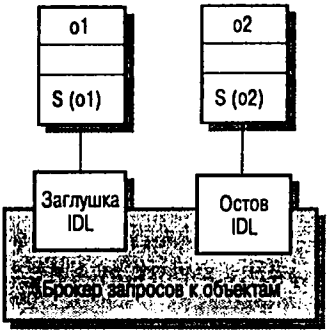


Рис. 11.12. Взаимодействие объектов посредством брокера запросов к объектам

Объект, предоставляющий сервис, связан с остовом (skeleton) IDL, который связывает интерфейс с реализацией сервисов. Иными словами, когда сервис вызывается через интерфейс, остов IDL транслирует вызов к сервису независимо от того, какой язык использовался в реализации. После завершения метода или процедуры остов транслирует результаты в язык IDL, так что они становятся доступными вызывающему объекту. Если объект одновременно предоставляет сервисы другим объектам или использует сервисы, которые предоставлены еще где-то, ему требуются и остов IDL, и заглушка IDL. Последняя необходима всем используемым объектам.

Брокер запросов к объектам обычно реализуется не в виде отдельных процессов, а как каркас (см. главу 14), который связан с реализацией объектов. Поэтому в распределенной системе каждый компьютер, на котором работают объекты, должен иметь собственный брокер запросов к объектам, который будет обрабатывать все локальные вызовы объектов. Но если запрос сделан к сервису, который предоставлен удаленным объектом, требуется взаимодействие между брокерами.

Такая ситуация проиллюстрирована на рис. 11.13. В данном примере, если объект o1 или o2 отправляет запросы к сервисам, предоставляемым объектами o3 или o4, то необходимо взаимодействие связанных с этими объектами брокеров. Стандарты CORBA поддерживают взаимодействие "брокер-брокер", которое обеспечивает брокерам доступ к описаниям интерфейсов IDL, и предлагают разработанный группой OMG стандарт обобщенного протокола взаимодействия брокеров GIOP (Generic Inter-ORB Protocol). Данный протокол определяет стандартные сообщения, которыми могут обмениваться брокеры при выполнении вызовов удаленного объекта и передаче информации. В сочетании с протоколом Internet низкого уровня TCP/IP этот протокол позволяет брокерам взаимодействовать через Internet.

Первые варианты CORBA были разработаны еще в 1980-х годах. Ранние версии CORBA просто были связаны с поддержкой распределенных объектов. Однако со временем стандарты развивались, становились более расширенными. Подобно механизмам

взаимодействия распределенных объектов, стандарты CORBA сейчас определяют некоторые стандартные сервисы, которые можно использовать для поддержки объектно-ориентированных приложений.

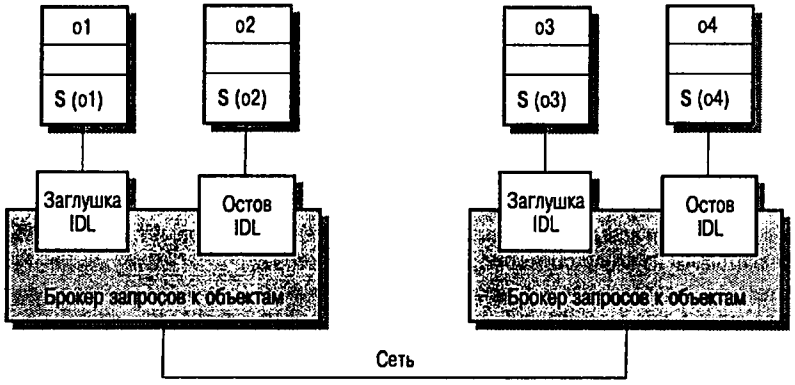


Рис. 11.13. Взаимодействие между брокерами запросов к объектам

Сервисы CORBA являются средствами, которые необходимы во многих распределенных системах. Эти стандарты определяют примерно 15 общих служб (сервисов). Вот некоторые из них.

1. Служба имен, которая позволяет объектам находить другие объекты в сети и ссылаться на них. Служба имен является сервисом каталогов, который присваивает имена объектам. При необходимости объекты через эту службу могут находить идентификаторы IOR других объектов.
2. Служба регистрации, которая позволяет объектам регистрировать другие объекты после совершения некоторых событий. С помощью этой службы объекты можно регистрировать по их участию в определенном событии, а когда данное событие уже произошло, оно автоматически регистрируется сервисом.
3. Служба транзакций, которая поддерживает элементарные транзакции и откат назад в случае ошибок или сбоев. Эта служба является отказоустойчивым средством (см. главу 18), обеспечивающим восстановление в случае ошибок во время операции обновления. Если действия по обновлению объекта приведут к ошибкам или сбою системы, данный объект всегда можно вернуть назад к тому состоянию, которое было перед началом обновления.

Считается, что стандарты CORBA должны содержать определения интерфейсов для широкого диапазона компонентов, которые могут использоваться при построении распределенных приложений. Эти компоненты могут быть вертикальными или горизонтальными. Вертикальные компоненты разрабатываются специально для конкретных приложений. Как уже отмечалось, разработкой определений этих компонентов занято множество специалистов из различных сфер деятельности. Горизонтальные компоненты универсальны, например компоненты пользовательского интерфейса.

Во время написания этой книги спецификации компонентов были уже разработаны, но еще не согласованы. С моей точки зрения, вероятно, именно здесь наиболее слабое место стандартов CORBA, и, возможно, потребуется несколько лет, чтобы достичь того, что в наличии будут и спецификации, и реализации компонентов.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Все современные большие системы в той или иной степени являются распределенными, в которых программные компоненты выполняются на интегрированной в сеть группе процессоров.
- Распределенным системам присущи следующие черты: совместное использование ресурсов, открытость, параллельность, масштабируемость, устойчивость к ошибкам и прозрачность.
- Системы клиент/сервер являются распределенными. Такие системы моделируются как набор сервисов, предоставляемых сервером клиентским процессам.
- В системе клиент/сервер интерфейс пользователя всегда запускается на стороне клиента, а управление данными всегда поддерживается на разделяемом сервере. Функции приложения могут быть реализованы на клиентском компьютере или на сервере.
- В архитектуре распределенных объектов нет различий между клиентами и серверами. Объекты предоставляют основные сервисы, которые могут вызывать другие объекты. Такой же подход можно использовать в реализации систем клиент/сервер.
- В системах распределенных объектов должно быть промежуточное программное обеспечение, предназначенное для обработки взаимодействий между объектами, а также добавления или удаления объектов из системы. Концептуально промежуточное ПО можно представить как программную шину, к которой подключены объекты.
- Стандарты CORBA представляют собой набор стандартов для промежуточного ПО, поддерживающего архитектуру распределенных объектов. К ним относятся определения модели объектов, брокера запросов к объектам и общих сервисов. В настоящее время существует несколько реализаций стандартов CORBA.

Упражнения

11.1. Объясните, почему распределенные системы всегда более масштабируемы, чем централизованные. Какой вероятный предел масштабируемости программных систем?

11.2. В чем основное отличие между моделями толстого и тонкого клиента в разработке систем клиент/сервер? Объясните, почему использование Java как языка реализации сглаживает различия между этими моделями?

11.3. На основе модели приложения, изображенной на рис. 11.4, рассмотрите возможные проблемы, которые могут возникнуть при преобразовании системы 1980-х годов, реализованной на мейнфрейме и предназначенной для работы в сфере здравоохранения, в систему архитектуры клиент/сервер.

11.4. Распределенные системы, базирующиеся на модели клиент/сервер, разрабатывались с 1980-х годов, но только недавно такие системы, основанные на распределенных объектах, были реализованы. Приведите три причины, почему так получилось.

11.5. Объясните, почему использование распределенных объектов совместно с брокером запросов к объектам упрощает реализацию масштабируемых систем клиент/сервер. Проиллюстрируйте свой ответ примером.

11.6. Каким образом используется язык IDL для поддержки взаимодействия между объектами, реализованными на разных языках программирования? Объясните, почему такой подход может вызвать проблемы, связанные с производительностью, если между языками, которые используются при реализации объектов, имеются радикальные различия.

11.7. Какие базовые средства должен предоставлять брокер запросов к объектам?

11.8. Можно показать, что разработка стандартов CORBA для горизонтальных и вертикальных компонентов ограничивает конкуренцию. Если они уже созданы и адаптированы, это препятствует разработке лучших компонентов более мелкими компаниями. Обсудите роль стандартизации в поддержке или ограничении конкуренции на рынке программного обеспечения.

12

Объектно-ориентированное проектирование

Цели

Цель настоящей главы — познакомить с подходом к проектированию программного обеспечения, в котором система представляется в виде взаимодействующих объектов. Прочитав эту главу, вы должны:

- знать, что структуру программы можно представить в виде совокупности взаимодействующих объектов, управляющих собственным состоянием и операциями;
- иметь представление об основных этапах процесса объектно-ориентированного проектирования;
- понимать различные модели, которые используются при документировании объектно-ориентированной структуры;
- познакомиться с представлением этих моделей с помощью UML.

Содержание

Объектно-ориентированное проектирование представляет собой стратегию, в рамках которой разработчики системы вместо операций и функций мыслят в понятиях *объекты*. Программная система состоит из взаимодействующих объектов, которые имеют собственное локальное состояние и могут выполнять определенный набор операций, определяемый состоянием объекта (рис. 12.1). Объекты скрывают информацию о представлении состояний и, следовательно, ограничивают к ним доступ. Под процессом объектно-ориентированного проектирования подразумевается проектирование классов объектов и взаимоотношений между этими классами. Когда проект реализован в виде исполняемой программы, все необходимые объекты создаются динамически с помощью определений классов.

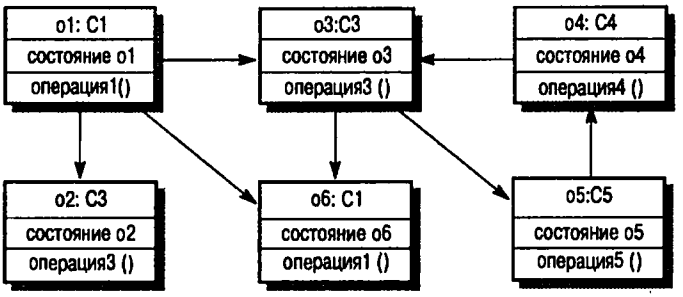


Рис. 12.1. Система взаимодействующих объектов

Объектно-ориентированное проектирование — только часть *объектно-ориентированного процесса разработки системы*, где на протяжении всего процесса создания ПО используется объектно-ориентированный подход. Этот подход подразумевает выполнение трех этапов.

- *Объектно-ориентированный анализ.* Создание объектно-ориентированной модели предметной области приложения ПО. Здесь объекты отражают реальные объекты-сущности, также определяются операции, выполняемые объектами.
- *Объектно-ориентированное проектирование.* Разработка объектно-ориентированной модели системы ПО (системной архитектуры) с учетом системных требований. В объектно-ориентированной модели определение всех объектов подчинено решению конкретной задачи.
- *Объектно-ориентированное программирование.* Реализация архитектуры (модели) системы с помощью объектно-ориентированного языка программирования. Такие языки, например Java, непосредственно выполняют реализацию определенных объектов и предоставляют средства для определения классов объектов.

Данные этапы могут "перетекать" друг в друга, т.е. могут не иметь четких рамок, причем на каждом этапе обычно применяется одна и та же система нотации. Переход на следующий этап приводит к усовершенствованию результатов предыдущего этапа путем более детального описания определенных ранее классов объектов и определения новых классов. Так как данные скрыты внутри объектов, детальные решения о представлении данных можно отложить до этапа реализации системы. В некоторых случаях можно также не спешить с принятием решений о расположении объектов и о том, будут ли эти объекты последовательными или параллельными. Все сказанное означает, что разработчики ПО не стеснены деталями реализации системы.

Объектно-ориентированные системы можно рассматривать как совокупность автономных и в определенной мере независимых объектов. Изменение реализации какого-нибудь объекта или добавление новых функций не влияет на другие объекты системы. Часто существует четкое соответствие между реальными объектами (например, аппаратными средствами) и управляющими ими объектами программной системы. Такой подход облегчает понимание и реализацию проекта.

Потенциально все объекты являются повторно используемыми компонентами, так как они независимо инкапсулируют данные о состоянии и операции. Архитектуру ПО можно разрабатывать на базе объектов, уже созданных в предыдущих проектах. Такой подход снижает стоимость проектирования, программирования и тестирования ПО. Кроме того, появляется возможность использовать стандартные объекты, что уменьшает риск, связанный с разработкой программного обеспечения. Однако, как показано в главе 14, иногда повторное использование эффективнее всего реализовать с помощью коллекций объектов (компонентов или объектных структур), а не через отдельные объекты.

В книгах [74, 295, 186, 54, 137, 13*, 32*, 34*] предлагаются различные методы объектно-ориентированного проектирования. В этих методах на протяжении всего процесса проектирования используется единообразная нотация, принятая в UML [304]. В данной главе не предлагаются какие-либо особые методы проектирования, а рассматриваются лишь общие концепции объектно-ориентированного проектирования. В разделе 12.2 рассмотрены этапы процесса проектирования. По всей главе используется система обозначений, принятая в UML.

12.1. Объекты и классы объектов

В настоящее время широко используются понятия *объект* и *объектно-ориентированный*. Эти термины применяются к различным типам объектов, методам проектирования, системам и языкам программирования. Во всех случаях применяется общее правило, согласно которому объект инкапсулирует данные о своем внутреннем строении. Это правило отражено в моем определении объекта и класса объектов.

> Объект — это нечто, способное пребывать в различных состояниях и имеющее определенное множество операций. Состояние определяется как набор атрибутов объекта. Операции, связанные с объектом, предоставляют сервисы (функциональные возможности) другим объектам (клиентам) для выполнения определенных вычислений.
>
> Объекты создаются в соответствии с определением класса объектов, которое служит шаблоном для создания объектов. В него включены объявления всех атрибутов и операций, связанных с объектом данного класса.

Нотация, которая используется здесь для обозначения классов объектов, определена в UML. Класс объектов представляется как прямоугольник с названием класса, разделенный на две секции. В верхней секции перечислены атрибуты объектов. Операции, связанные с данным объектом, расположены в нижней секции. Пример такой нотации представлен на рис. 12.2, где показан класс объектов, моделирующий служащего некой организации. В UML термин *операция* является спецификацией некоторого действия, а термин *метод* обычно относится к реализации данной операции.

Класс Работник определяется рядом атрибутов, в которых содержатся данные о служащих, в том числе их имена и адрес, коды социального обеспечения, налоговые коды и т.д. Конечно, на самом деле атрибутов, ассоциированных с классом, больше, чем изображено на рисунке. Определены также операции, связанные с объектами: принять (выпол-

няется при поступлении на работу), уволить (выполняется при увольнении служащего из организации), пенсия (выполняется, если служащий становится пенсионером организации) и изменитьДанные (выполняется в случаях, если требуется внести изменения в имеющиеся данные о работнике).

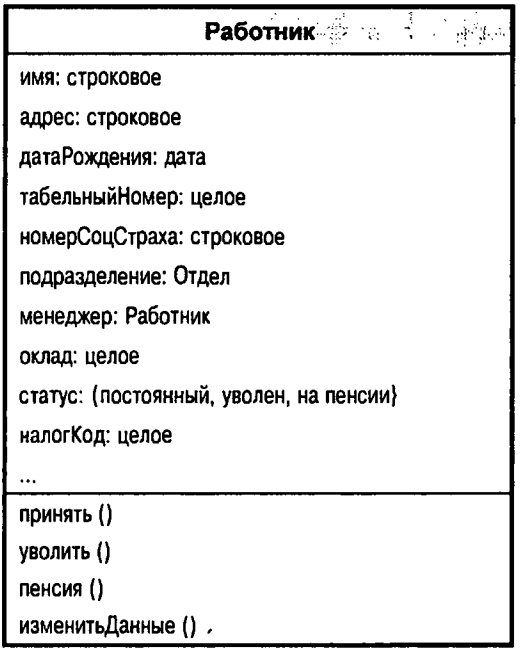


Рис. 12.2. Объект Работник

Взаимодействие между объектами осуществляется посредством запросов к сервисам (вызов методов) из других объектов и при необходимости путем обмена данными, требующимися для поддержки сервиса. Копии данных, необходимых для работы сервиса, и результаты работы сервиса передаются как параметры. Вот несколько примеров такого стиля взаимодействия.

```
// Вызов метода, ассоциированного с объектом Buffer (Буфер),
//который возвращает следующее значение в буфер
v = circularBuffer.Get ();
// Вызов метода, связанного с объектом thermostat (термостат),
//который поддерживает нужную температуру
thermostat.setTemp (20);
```

В некоторых распределенных системах взаимодействие между объектами реализовано непосредственно в виде текстовых сообщений, которыми обмениваются объекты. Объект, получивший сообщение, выполняет его грамматический разбор, идентифицирует сервис и связанные с ним данные и запускает запрашиваемый сервис. Однако, если объекты сосуществуют в одной программе, вызовы методов реализованы аналогично вызовам процедур или функций в языках программирования, например таких, как C или Ada.

Если запросы к сервису реализованы именно таким образом, взаимодействие между объектами синхронно. Это означает, что объект, отправивший запрос к сервису, ожидает окончания выполнения запроса. Однако, если объекты реализованы как параллельные

процессы или потоки, взаимодействие объектов может быть асинхронным. Отправив запрос к сервису, объект может продолжить работу и не ждать, пока сервис выполнит его запрос. Ниже в этом разделе показано, каким образом можно реализовать объекты как параллельные процессы.

Как отмечалось в главе 7, в которой описан ряд объектных моделей, классы объектов можно упорядочить или в виде иерархии обобщения или в виде иерархии наследования, которые показывают отношения между основными и частными классами объектов. Эти частные классы объектов полностью совместимы с основными классами, но содержат больше информации. В системе обозначений UML направление обобщения указывается стрелками, направленными на родительский класс. В объектно-ориентированных языках программирования обобщение обычно реализуется через механизм наследования. Производный класс (класс-потомок) наследует атрибуты и операции от родительского класса.

Пример такой иерархии изображен на рис. 12.3, где показаны различные классы работников. Классы, расположенные внизу иерархии, имеют те же атрибуты и операции, что и родительские классы, но могут содержать новые атрибуты и операции или же изменять имеющиеся в родительских классах. Если в модели используется имя родительского класса, значит, объект в системе может быть определен либо самим классом, либо любым из его потомков.

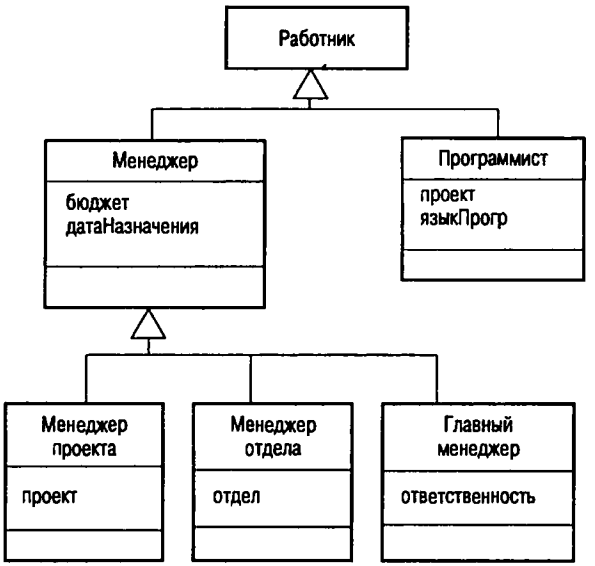


Рис. 12.3. Иерархия обобщения

На рис. 12.3 видно, что класс **Менеджер** обладает всеми атрибутами и операциями класса **Работник** и, кроме того, имеет два новых атрибута: ресурсы, которыми управляет менеджер (**бюджет**), и дата назначения его на должность менеджера (**датаНазначения**). Также добавлены новые атрибуты в класс **Программист**. Один из них определяет проект, над которым работает программист, другой характеризует уровень его профессионализма при использовании определенного языка программирования (**языкПрогр**). Таким образом, объекты класса **Менеджер** и **Программист** можно использовать вместо объектов класса **Работник**.

Объекты, являющиеся членами класса объектов, взаимодействуют с другими объектами. Эти взаимоотношения моделируются с помощью описания связей (ассоциаций) между классами объектов. В UML связь обозначается линией, которая соединяет классы объектов, причем линия может быть снабжена информацией о данной связи. На рис. 12.4 показаны связи между объектами классов Работник и Отдел и между объектами классов Работник и Менеджер.

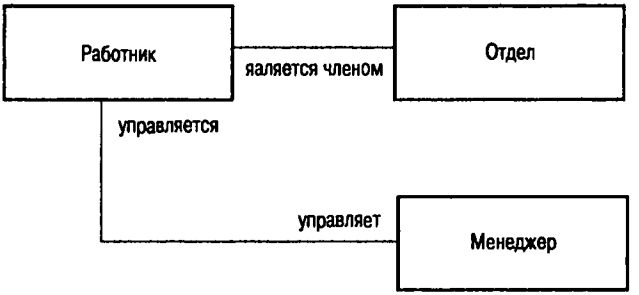


Рис. 12.4. Модель связей

Связи представляют самые общие отношения и часто используются в UML там, где требуется указать, что какое-то свойство объекта является связанным с объектом или же реализация метода объекта полагается на связанный объект. Однако в принципе тип связи может быть каким угодно. Одним из наиболее распространенных типов связи, который служит для создания новых объектов из уже имеющихся, является агрегирование. Этот тип связи рассмотрен в главе 7.

12.1.1. Параллельные объекты

В общем случае объекты запрашивают сервис от любого объекта посредством передачи ему сообщения "запрос к сервису". Обычно нет необходимости в последовательном выполнении, при котором один объект ожидает завершения работы сервиса по сделанному запросу. Общая модель взаимодействия объектов позволяет их одновременное выполнение в виде параллельных процессов. Такие объекты могут выполняться на одном компьютере или на разных машинах как распределенные объекты.

На практике в большинстве объектно-ориентированных языков программирования по умолчанию реализована модель последовательного выполнения, в которой запросы к сервисам объектов и вызовы функций реализованы одним и тем же способом. Например, на языке Java, когда объект, вызвавший объект theList (Список), создается из обычного класса объектов, это запишется так:

```
theList.append(17)
```

Здесь вызывается метод append (добавить), связанный с объектом theList, который добавляет элемент 17 в список theList, а выполнение объекта, сделавшего вызов, приостанавливается до тех пор, пока не завершится операция добавления. Однако в Java существует очень простой механизм потоков (threads), который позволяет создавать параллельно выполняющиеся объекты. Поэтому объектно-ориентированную архитектуру программной системы можно преобразовать так, чтобы объекты стали параллельными процессами.

Существует два типа параллельных объектов.

1. *Серверы*, в которых объект реализован как параллельный процесс с методами, соответствующими определенным операциям объекта. Методы запускаются в ответ на внешнее сообщение и могут выполняться параллельно с методами, связанными с другими объектами. По окончании всех действий выполнение объекта приостанавливается и он ожидает дальнейших запросов к сервису.

2. *Активные объекты*, у которых состояние может изменяться посредством операций, выполняющихся внутри самого объекта. Процесс, представляющий объект, постоянно выполняет эти операции, а следовательно, никогда не останавливается.

Серверы наиболее полезны в распределенных средах, где вызывающий и вызываемый объекты выполняются на разных компьютерах. Время ответа, которое требуется сервису, заранее не известно, поэтому где только можно следует спроектировать систему так, чтобы объект, отправивший запрос к сервису, не ждал, пока сервис выполнит запрос. Также серверы могут использоваться на одной машине, где им требуется некоторое время для выполнения запроса (например, печать документа) и где есть вероятность отправки запросов к сервису от нескольких разных объектов.

Активные объекты используются там, где объектам необходимо обновлять свое состояние через определенные интервалы времени. Такие объекты характерны для систем реального времени, в которых объекты связаны с аппаратными устройствами, собирающими информацию из окружения среды. Методы объектов позволяют другим объектам получить доступ к информации, определяющей состояние объекта.

В листинге 12.1 показано, как на языке Java можно определить и реализовать активный объект. Данный класс объектов представляет бортовой радиомаяк-ответчик (transponder) самолета. С помощью спутниковой навигационной системы радиомаяк-ответчик отслеживает положение самолета. Он может отвечать на сообщения, приходящие от компьютеров, управляющих воздушными полетами. В ответ на запрос метод givePosition сообщает текущее положение самолета.

Листинг 12.1. Реализация активного объекта, использующего потоки языка Java

```
class Transponder extends Thread {
       Position currentPosition ;
       Coords c1, c2 ;
       Satellite sat1, sat2 ;
       Navigator theNavigator ;

       public Position givePosition ()
       {
           return currentPosition;
       }
       public void run ()
       {
           while (true)
           {
               c1 = sat1.position ();
               c2 = sat2.position ();
               currentPosition = theNavigator.compute (c1, c2) ;
           }
       }
}//Transponder
```

Данный объект реализован как поток, где в непрерывном цикле метода run содержится код, вычисляющий положение самолета с помощью сигналов, полученных от спутников. В Java потоки создаются с помощью встроенного класса Thread (Поток), выступающего в объявлении классов в качестве базового.

12.2. Процесс объектно-ориентированного проектирования

В этом разделе процесс объектно-ориентированного проектирования показан на примере разработки структуры управляющей программной системы, встроенной в автоматизированную метеостанцию. Как отмечалось выше, есть несколько методов объектно-ориентированного проектирования, причем какого-либо предпочтительного метода или процесса проектирования не существует. Рассматриваемый здесь процесс является достаточно общим, т.е. состоит из операций, характерных для большинства процессов объектно-ориентированного проектирования. В этом отношении он сравним с процессом, предлагаемым языком UML [304], однако я значительно упростил его.

Общий процесс объектно-ориентированного проектирования состоит из нескольких этапов.

1. Определение рабочего окружения системы и разработка моделей ее использования.
2. Проектирование архитектуры системы.
3. Определение основных объектов системы.
4. Разработка моделей архитектуры системы.
5. Определение интерфейсов объекта.

Процесс проектирования нельзя представить в виде простой схемы, в которой предполагается четкая последовательность этапов. Фактически все перечисленные этапы в значительной мере можно выполнять параллельно, с взаимным влиянием друг на друга. Как только разработана архитектура системы, определяются объекты и (частично или полностью) интерфейсы. После создания моделей объектов отдельные объекты можно переопределить, а это может привести к изменениям в архитектуре системы. Далее в этом разделе каждый этап процесса проектирования обсуждается отдельно.

Пример ПО, которым я воспользуюсь для иллюстрации объектно-ориентированного проектирования, представляет собой часть системы, создающей метеорологические карты на основе автоматически собранных метеорологических данных. Подробное перечисление требований для такой системы займет много страниц. Однако, даже ограничившись кратким описанием системы, можно разработать ее общую архитектуру.

> Одним из требований системы построения карты погоды является регулярное обновление метеорологических карт на основе данных, полученных от удаленных метеостанций и других источников, например наблюдателей, метеозондов и спутников. В ответ на запрос регионального компьютера системы обслуживания метеостанций передают ему свои данные.
>
> Региональная компьютерная система объединяет данные из различных источников. Собранные данные архивируются и с помощью данных из этого архива и базы данных цифровых карт создается набор локальных метеорологических карт. Карты можно распечатать, направив их на специальный принтер, или же отобразить в разных форматах.

Из данного описания видно, что одна часть общей системы занимается сбором данных, другая обобщает данные, полученные из различных источников, третья выполняет архивирование данных и наконец четвертая создает метеорологические карты. На рис. 12.5 изображена одна из возможных архитектур системы, которую можно построить на основе предложенного описания. Она представляет собой многоуровневую архитектуру (обсуждаемую в главе 10), в которой отражены все этапы обработки данных в системе, т.е. сбор данных, обобщение данных, архивирование данных и создание карт. Такая многоуровневая архитектура вполне годится для нашей системы, так как каждый этап основывается только на обработке данных, выполненной на предыдущем этапе.

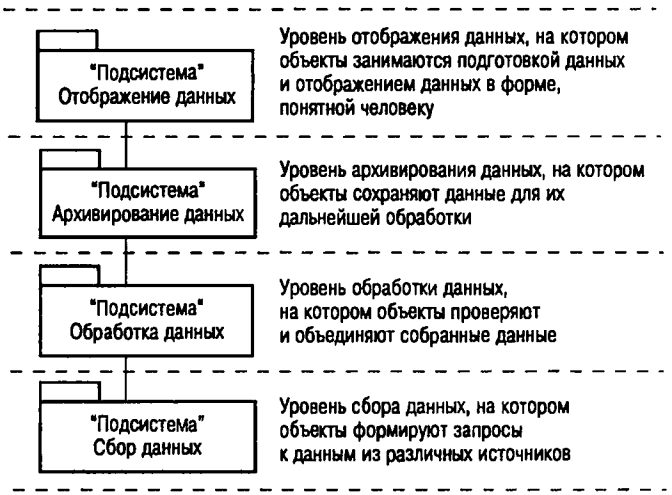


Рис. 12.5. Многоуровневая архитектура системы построения карт погоды

На рис. 12.5 показаны все уровни системы. Названия уровней заключены в прямоугольники, что в нотации UML обозначает подсистемы. Прямоугольники UML (т.е. подсистемы) – это набор объектов и других подсистем. Я использую здесь это обозначение, чтобы показать, что каждый уровень включает в себя множество других компонентов.

На рис. 12.6 изображена расширенная модель архитектуры, в которой показаны компоненты подсистем. Эти компоненты также очень абстрактны и построены на информации, содержащейся в описании системы. Продолжим рассматривать этот пример, уделяя особое внимание подсистеме **Метеостанция**, которая является частью уровня **Сбор данных**.

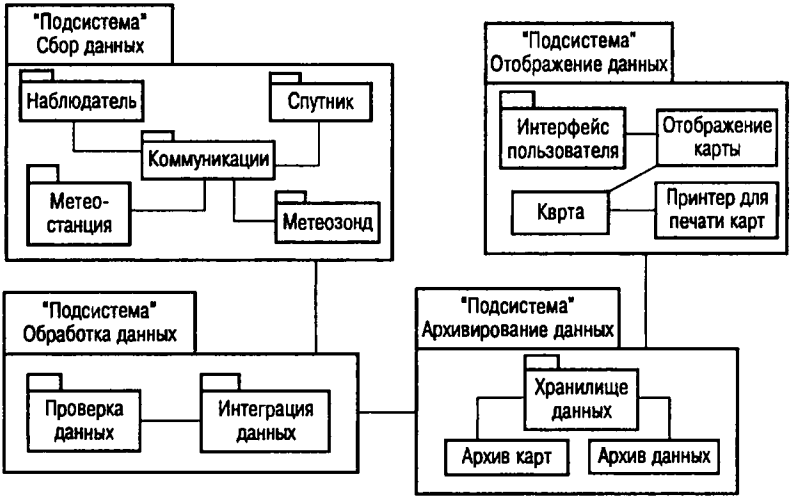


Рис. 12.6. Подсистемы в системе построения карт погоды

12.2.1. Окружение системы и модели ее использования

Первый этап в любом процессе проектирования состоит в выявлении взаимоотношений между проектируемым программным обеспечением и его окружением. Выявление этих взаимоотношений помогает решить, как обеспечить необходимую функциональность системы и как структурировать систему, чтобы она могла эффективно взаимодействовать со своим окружением.

Модель окружения системы и модель использования системы представляют собой две дополняющие друг друга модели взаимоотношений между данной системой и ее окружением.

1. Модель окружения системы — это статическая модель, которая описывает другие системы из окружения разрабатываемого ПО.
2. Модель использования системы — динамическая модель, которая показывает взаимодействие данной системы со своим окружением.

Модель окружения системы можно представить с помощью схемы связей (см. рис. 12.4), которая дает простую блок-схему общей архитектуры системы. С помощью *пакетов* языка UML ее можно представить в развернутом виде как совокупность подсистем (см. рис. 12.6). Такое представление показывает, что рабочее окружение системы Метеостанция находится внутри подсистемы, занимающейся сбором данных. Там же показаны другие подсистемы, которые образуют систему построения карт погоды.

При моделировании взаимодействия системы с ее окружением применяется абстрактный подход, который не требует больших объемов данных для описания этих взаимодействий. Подход, предлагаемый UML, состоит в том, чтобы разработать модель вариантов использования, в которой каждый вариант представляет собой определенное взаимодействие с системой (см. главу 6). В модели вариантов использования каждое возможное взаимодействие изображается в виде эллипса, а внешняя сущность, включенная во взаимодействие, представлена стилизованной фигуркой человека. В нашем примере внешняя сущность, хотя и представлена фигуркой человека, является системой обработки метеорологических данных.

Модель вариантов использования для метеостанции показана на рис. 12.7. В этой модели метеостанция взаимодействует с внешними объектами во время запуска и завершения работы, при составлении отчетов на основе собранных данных, а также при тестировании и калибровке метеорологических приборов.

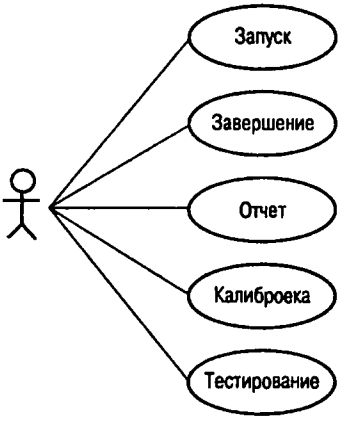


Рис. 12.7. Варианты использования метеостанции

Каждый из имеющихся вариантов использования можно описать с помощью простого естественного языка. Такое описание помогает разработчикам проекта идентифицировать объекты в системе и понять, что система должна делать. Я использую стилизованную форму описания, которая четко определяет, как происходит обмен информацией, как инициируется взаимодействие и т.д. Эта форма описания показана в табл. 12.1, где представлен вариант использования Отчет (см. рис. 12.7).

Таблица 12.1. Описание варианта использования Отчет

Система	Метеостанция
Вариант использования	Отчет
Участники	Система сбора метеорологических данных, метеостанция
Данные	Метеостанция отправляет сводку с данными, снятыми с различных приборов в определенный временной период системой сбора метеорологических данных. В сообщении содержатся максимальные, минимальные и средние значения температуры почвы и воздуха, атмосферного давления, скорости ветра, общее количество выпавших осадков и направление ветра, взятые через пятиминутные интервалы времени
Входные сигналы	Система сбора метеорологических данных устанавливает модемную связь с метеостанцией и отправляет запрос на передачу данных
Ответ	Итоговые данные отправляются в систему сбора метеорологических данных
Комментарии	Обычно от метеостанций запрашивают отчет каждый час, но эта частота запросов может отличаться для разных станций, а также может измениться в будущем

Конечно, для описания вариантов использования можно прибегнуть к любой другой методике при условии, что предложенное описание краткое и понятное. Как правило, требуется разработать описания для всех вариантов использования, имеющихся в данной модели.

Описание вариантов использования помогает идентифицировать объекты и операции в системе. Из описания варианта использования Отчет видно, что в системе должны быть объекты, представляющие приборы для сбора метеорологических данных, а также объекты, предоставляющие итоговые метеорологические данные. Должны также быть операции, формирующие запрос, и операции, пересылающие метеорологические данные.

12.2.2. Проектирование архитектуры

Когда взаимодействия между проектируемой системой ПО и ее окружением определены, эти данные можно использовать как основу для разработки архитектуры системы. Конечно, при этом необходимо применять знания об общих принципах проектирования системных архитектур и данные о конкретной предметной области.

Автоматизированная метеостанция является относительно простой системой, поэтому ее архитектуру можно вновь представить как многоуровневую модель. На рис. 12.8 внутри большого прямоугольника Метеостанция расположены три прямоугольника UML. Здесь я использовал систему нотации UML (текст в прямоугольниках с загнутыми углами) с тем, чтобы представить дополнительную информацию.

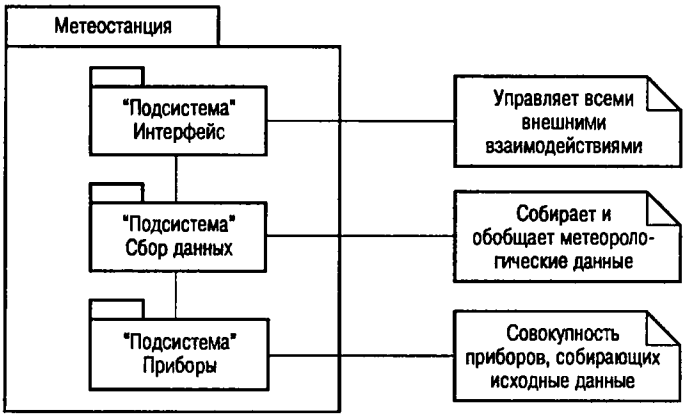


Рис. 12.8. Архитектура метеостанции

В программном обеспечении метеостанции можно выделить три уровня.

1. Уровень интерфейсов, который занимается всеми взаимодействиями с другими частями системы и предоставлением внешних интерфейсов системы.
2. Уровень сбора данных, управляющий сбором данных с приборов и обобщающий метеорологические данные перед отправкой их в систему построения карт погоды.
3. Уровень приборов, в котором представлены все приборы, используемые в процессе сбора исходных метеорологических данных.

В общем случае следует попытаться разложить систему на части так, чтобы архитектура была как можно проще. Согласно хорошему практическому правилу, модель архитектуры должна состоять не более чем из семи основных объектов. Каждый такой объект можно описать отдельно, однако для того, чтобы отобразить структуру этих объектов и их взаимосвязи, можно воспользоваться схемой, подобной показанной на рис. 12.6.

12.2.3. Определение объектов

Перед выполнением данного этапа проектирования уже должны быть сформированы представления относительно основных объектов проектируемой системы. В системе метеостанции очевидно, что приборы являются объектами и требуется по крайней мере один объект на каждом уровне архитектуры. Это проявление основного принципа, согласно которому объекты обычно появляются в процессе проектирования. Вместе с тем требуется определить и документировать все другие объекты системы.

Хотя этот раздел назван "Определение *объектов*", на самом деле на данном этапе проектирования определяются *классы* объектов. Структура системы описывается в терминах этих классов. Классы объектов, определенные ранее, неизбежно получают более детальное описание, поэтому иногда приходится возвращаться на данный этап проектирования для переопределения классов.

Существует множество подходов к определению классов объектов.

1. Использование грамматического анализа естественного языкового описания системы. Объекты и атрибуты — это существительные, операции и сервисы — глаголы [1]. Такой подход реализован в иерархическом методе объектно-ориентированного про-

ектирования [295], который широко используется в аэрокосмической промышленности Европы.

2. Использование в качестве объектов ПО событий, объектов и ситуаций реального мира из области приложения, например самолетов, ролевых ситуаций менеджера, взаимодействий, подобных интерактивному общению на научных конференциях и т.д. [313, 74, 343, 13*, 33*]. Для реализации таких объектов могут потребоваться специальные структуры хранения данных (абстрактные структуры данных).
3. Применение подхода, при котором разработчик сначала полностью определяет поведение системы. Затем определяются компоненты системы, отвечающие за различные поведенческие акты (режимы работы системы), при этом основное внимание уделяется тому, кто инициирует и кто осуществляет данные режимы. Компоненты системы, отвечающие за основные режимы работы, считаются объектами [301].
4. Применение подхода, основанного на сценариях, в котором по очереди определяются и анализируются различные сценарии использования системы. Поскольку анализируется каждый сценарий, группа, отвечающая за анализ, должна идентифицировать необходимые объекты, атрибуты и операции. Метод анализа, при котором аналитики и разработчики присваивают роли объектам, показывает эффективность подхода, основанного на сценариях [33].

Каждый из описанных подходов помогает начать процесс определения объектов. Но для описания объектов и классов объектов необходимо использовать информацию, полученную из разных источников. Объекты и операции, первоначально определенные на основе неформального описания системы, вполне могут послужить отправной точкой при проектировании. Затем для усовершенствования и расширения описания первоначальных объектов можно использовать дополнительную информацию, полученную из области применения ПО или анализа сценариев. Дополнительную информацию также можно получить в ходе обсуждения с пользователями разрабатываемой системы или анализа имеющихся систем.

При определении объектов метеостанции я использую смешанный подход. Чтобы описать все объекты, потребуется много места, поэтому на рис. 12.9 я показал только пять классов объектов. ПочвенныйТермометр, Анемометр и Барометр являются объектами области приложения, а объекты Метеостанция и МетеоДанные определены на основе описания системы и описания сценариев (вариантов использования).

Все объекты связаны с различными уровнями в архитектуре системы.

1. Класс объектов Метеостанция предоставляет основной интерфейс метеостанции при работе с внешним окружением. Поэтому операции класса соответствуют взаимодействиям, показанным на рис. 12.7. В данном случае, чтобы описать все эти взаимодействия, я использую один класс объектов, но в других проектах для представления интерфейса системы, возможно, потребуется использовать несколько классов.
2. Класс объектов МетеоДанные инкапсулирует итоговые данные от различных приборов метеостанции. Связанные с ним операции собирают и обобщают данные.
3. Классы объектов ПочвенныйТермометр, Анемометр и Барометр отображают реальные аппаратные средства метеостанции, соответствующие операции этих классов должны управлять данными приборами.

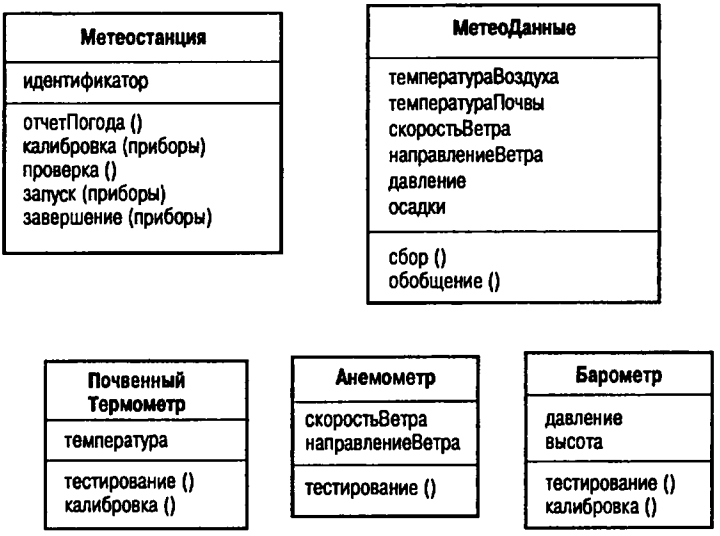


Рис. 12.9. Примеры классов объектов системы метеостанции

На этом этапе проектирования знания из области приложения ПО можно использовать для идентификации будущих объектов и сервисов. В нашем примере известно, что метеостанции обычно расположены в удаленных местах. Они оснащены различными приборами, которые иногда дают сбой в работе. Отчет о неполадках в приборах должен отправляться автоматически. А это значит, что необходимы атрибуты и операции, которые проверяли бы правильность функционирования приборов. Кроме того, необходимо идентифицировать данные, собранные со всех станций; таким образом, каждая метеостанция должна иметь собственный идентификатор.

Я решил не создавать объекты, ассоциированные с активными объектами каждого прибора. Чтобы снять данные в нужное время, объекты приборов вызывают операцию **сбор** объекта **МетеоДанные**. Активные объекты имеют собственное управление, в нашем примере предполагается, что каждый прибор сам определяет, когда нужно проводить замеры. Однако такой подход имеет недостаток: если принято решение изменить временной интервал сбора данных или если разные метеостанции собирают данные через разные промежутки времени, тогда необходимо вводить новые классы объектов. Создавая объекты приборов, снимающие показания по запросу, любые изменения в стратегии сбора данных можно легко реализовать без изменения объектов, связанных с приборами.

12.2.4. Модели архитектуры

Модели системной архитектуры показывают объекты и классы объектов, составляющих систему, и при необходимости типы взаимоотношений между этими объектами. Такие модели служат мостом между требованиями к системе и ее реализацией. А это значит, что к данным моделям предъявлены противоречивые требования. Они должны быть абстрактными настолько, чтобы лишние данные не скрывали отношения между моделью архитектуры и требованиями к системе. Однако, чтобы программист мог принимать решения по реализации, модель должна содержать достаточное количество информации.

Эти противоречие можно обойти, разработав несколько моделей разного уровня детализации. Там, где существуют тесные рабочие связи между разработчиками требований, проектировщиками и программистами, можно обойтись одной обобщенной моделью. В этом

случае конкретные решения по архитектуре системы можно принимать в процессе реализации системы. Когда связи между разработчиками требований, проектировщиками и программистами не такие тесные (например, если система проектируется в одном подразделении организации, а реализуется в другом), требуется более детализированная модель.

Следовательно, в процессе проектирования важно решить, какие требуются модели и какой должна быть степень их детализации. Это решение зависит также от типа разрабатываемой системы. Систему последовательной обработки данных можно спроектировать на основе встроенной системы реального времени разными способами с использованием различных моделей архитектуры. Существует множество систем, которым требуются все виды моделей. Но уменьшение числа созданных моделей сокращает расходы и время проектирования.

Существует два типа объектно-ориентированных моделей системной архитектуры.

1. Статические модели, которые описывают статическую структуру системы в терминах классов объектов и взаимоотношений между ними. Основными взаимоотношениями, которые документируются на данном этапе, являются отношения обобщения, отношения "используют–используются" и структурные отношения.
2. Динамические модели, которые описывают динамическую структуру системы и показывают взаимодействия между объектами системы (но не классами объектов). Документируемые взаимодействия содержат последовательность составленных объектами запросов к сервисам и описывают реакцию системы на взаимодействия между объектами.

В языке моделирования UML поддерживается огромное количество возможных статических и динамических моделей. Буч (Booch, [55]) предлагает девять различных типов схем для представления моделей. Чтобы показать все модели, не хватит места, да и не все из них пригодны для примера с метеостанцией. Здесь рассматриваются три типа моделей.

1. Модели подсистем, которые показывают логически сгруппированные объекты. Они представлены с помощью диаграммы классов, в которой каждая подсистема обозначается как пакет. Модели подсистем являются статическими.
2. Модели последовательностей, которые показывают последовательность взаимодействий между объектами. Они представляются в UML с помощью диаграмм последовательности или кооперативных диаграмм. Это динамические модели.
3. Модели конечного автомата, которые показывают изменение состояния отдельных объектов в ответ на определенные события. В UML они представлены в виде диаграмм состояния. Модели конечного автомата являются динамическими.

Другие типы моделей рассмотрены ранее в этой и предыдущих главах. Модели вариантов использования показывают взаимодействия с системой (см. рис. 12.7, 6.11 и 6.12), модели объектов дают описание классов объектов (см. рис. 12.2), модели обобщения и наследования (см. рис. 7.8–7.10) показывают, какие классы являются обобщениями других классов, модель агрегирования (см. рис. 7.11) выявляет взаимосвязи между коллекциями объектов.

С моей точки зрения, модель подсистем является одной из наиболее важных и полезных статических моделей, поскольку показывает, как можно организовать систему в виде логически связанных групп объектов. Мы уже встречали примеры такого типа модели на рис. 12.6, где изображены подсистемы системы построения карт погоды. В UML пакеты являются структурами инкапсуляции и не отображаются непосредственно в объектах разрабатываемой системы. Однако они могут отображаться, например, в виде библиотек Java.

На рис. 12.10 показаны объекты подсистем метеостанции. В данной модели также представлены некоторые связи. Например, объект **КонтроллерКоммуникаций** связан с объектом **Метеостанция**, а объект **Метеостанция** связан с пакетом **Сбор данных**. Совместная модель пакетов и классов объектов позволяет показать логически сгруппированные системные элементы.

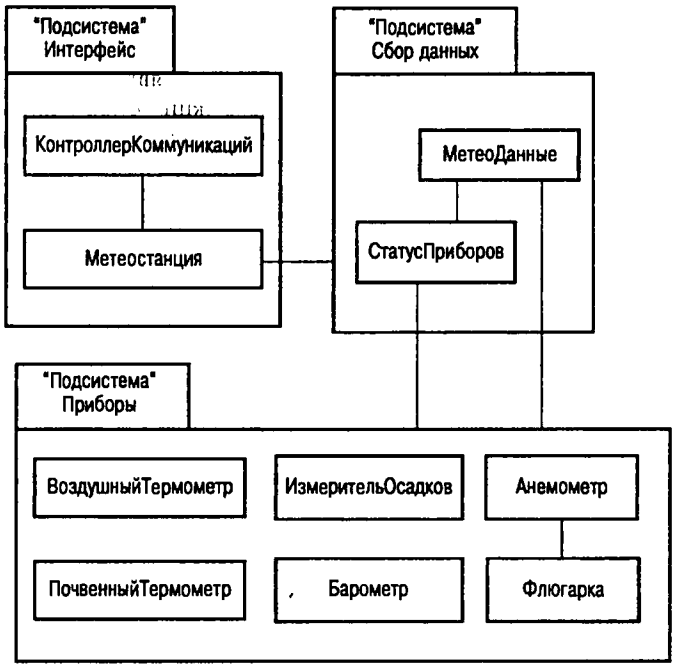


Рис. 12.10. Пакеты системы метеостанции

Модель последовательностей — одна из наиболее полезных и наглядных динамических моделей, которая в каждом узле взаимодействия документирует последовательность происходящих между объектами взаимодействий. Опишем основные свойства модели последовательности.

1. Объекты, участвующие во взаимодействии, располагаются горизонтально вверху диаграммы. От каждого объекта исходит пунктирная вертикальная линия — линия жизни объекта.
2. Время направлено сверху вниз по пунктирным вертикальным линиям. Поэтому в данной модели легко увидеть последовательность операций.
3. Взаимодействия между объектами представлены маркированными стрелками, связывающими вертикальные линии. Это не поток данных, а представление сообщений или событий, основных в данном взаимодействии.
4. Тонкий прямоугольник на линии жизни объекта обозначает интервал времени, в течение которого данный объект был управляющим объектом системы. Объект берет на себя управление в верхней части прямоугольника и передает управление другому объекту внизу прямоугольника. Если в системе имеется иерархия вызовов, то управление не передается до тех пор, пока не завершится последний возврат в вызове первоначального метода.

Сказанное выше проиллюстрировано на рис. 12.11, где изображена последовательность взаимодействий в тот момент, когда внешняя система посылает метеостанции запрос на получение данных. Диаграмму можно прокомментировать следующим образом.

1. Объект :КонтроллерКоммуникаций, являющийся экземпляром одноименного класса, получает внешний запрос "отправить отчет о погоде". Он подтверждает получение запроса. Половинная стрелка показывает, что, отправив сообщение, объект не ожидает ответа.
2. Этот объект отправляет сообщение объекту, который является экземпляром класса Метеостанция, чтобы создать метеорологический отчет. Объект :КонтроллерКоммуникаций затем приостанавливает работу (его прямоугольник управления заканчивается). Используемый стиль стрелок показывает, что объекты :КонтроллерКоммуникаций и :Метеостанция могут выполняться параллельно.
3. Объект, который является экземпляром класса Метеостанция, отправляет сообщение объекту :МетеоДанные, чтобы подвести итоги по метеорологическим данным. Здесь другой стиль стрелок указывает на то, что объект :Метеостанция ожидает ответа.
4. После составления сводки, управление передается объекту :Метеостанция. Пунктирная стрелка обозначает возврат управления.
5. Этот объект передает сообщение объекту :КонтроллерКоммуникаций, из которого был прислан запрос, чтобы передать данные в удаленную систему. Затем объект :Метеостанция приостанавливает работу.
6. Объект :КонтроллерКоммуникаций передает сводные данные в удаленную систему, получает подтверждение и затем переходит в состояние ожидания следующего запроса.

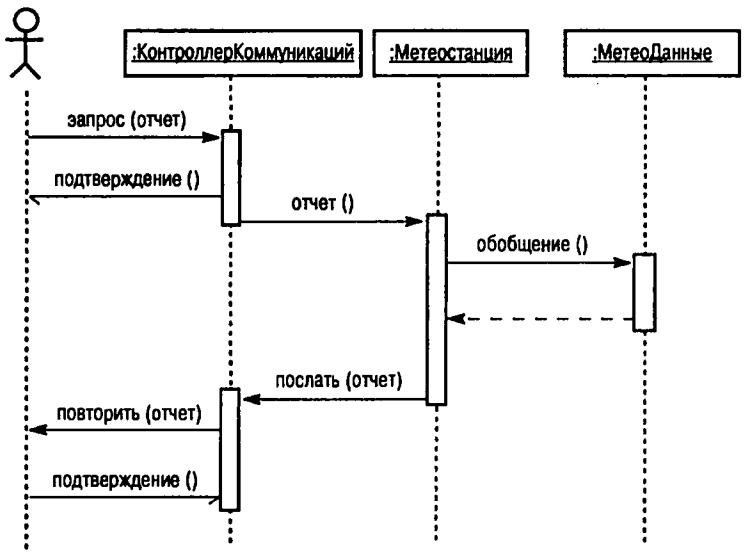


Рис. 12.11. Последовательность операций во время сбора данных

Из диаграммы последовательностей видно, что объекты КонтроллерКоммуникаций и Метеостанция в действительности являются параллельными процессами, выполнение которых может приостанавливаться и снова возобновляться. Здесь существенно, что экзем-

пляр объекта КонтроллерКоммуникаций получает сообщения от внешней системы, расшифровывает полученные сообщения и инициализирует действия метеостанции.

При документировании проекта для каждого значительного взаимодействия необходимо создавать диаграмму последовательностей. Если разрабатывается модель вариантов использования, то диаграмму последовательности нужно создавать для каждого заданного варианта.

Диаграммы последовательностей обычно применяются при моделировании комбинированного поведения групп объектов, однако при желании можно также показать поведение одного объекта в ответ на обрабатываемые им сообщения. В UML для описания моделей конечного автомата используются диаграммы состояний.

На рис. 12.12 представлена диаграмма состояния объекта Метеостанция, которая показывает реакцию объекта на запросы от разных сервисов.

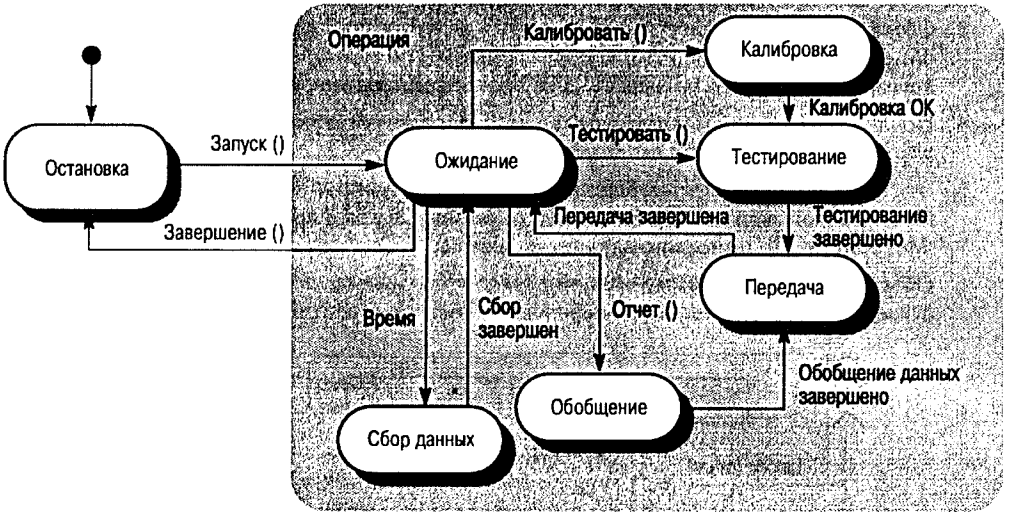


Рис. 12.12. Диаграмма состояний объекта Метеостанция

Эту диаграмму можно прокомментировать следующим образом.

1. Если объект находится в состоянии Останов, он может отреагировать только сообщением запуск(). Затем он переходит в состояние ожидания дальнейших сообщений. Немаркированная стрелка с черным кружком указывает на то, что это состояние является начальным.
2. В состоянии Ожидание система ожидает дальнейших сообщений. При получении сообщения завершение() объект возвращается в состояние завершения работы Останов.
3. Если получено сообщение отчет(), то система переходит в состояние обобщения данных Обобщение, а затем в состояние передачи данных Передача, в котором информация передается через объект КонтроллерКоммуникаций. Затем система возвращается в состояние ожидания.
4. Получив сообщение калибровать(), система последовательно проходит через состояния калибровки, тестирования, передачи и лишь после этого переходит в состояние ожидания. В случае получения сообщения тестировать(), система сразу переходит в состояние тестирования.

5. Если получен сигнал **время**, система переходит в состояние сбора данных, в котором она собирает данные от приборов. Каждый прибор по очереди также получает инструкцию "снять свои данные".

Обычно не нужно создавать диаграммы состояний для всех определенных в системе объектов. Большинство объектов относительно просты, и модель конечного автомата просто излишняя.

12.2.5. Специфицирование интерфейсов объектов

Важной частью любого процесса проектирования является специфицирование интерфейсов между различными компонентами системы. Интерфейсы необходимо определить так, чтобы объекты и другие компоненты можно было проектировать параллельно. Определив интерфейс, разработчики других объектов могут считать, что интерфейс уже реализован.

Одному объекту не обязательно должен соответствовать один интерфейс. Один и тот же объект может иметь несколько интерфейсов, причем каждый из них предлагает свой способ поддержки методов. Такая поддержка имеется непосредственно в Java, где интерфейсы объявляются отдельно от объектов и объекты "реализуют" интерфейсы. Другими словами, через один интерфейс можно получить доступ к набору объектов.

Проектирование интерфейсов объектов связано со спецификацией интерфейса в объекте или группе объектов. Под этим подразумевается определение сигнатур и семантик сервисов, которые поддерживаются этим объектом или группой объектов. В UML интерфейсы можно определить подобно диаграмме классов. Однако раздела свойств там нет, поэтому в стандарте UML шаблон <<интерфейс>> следует включать в именную часть.

Я предпочитаю альтернативный подход, в котором при определении интерфейса применяется язык программирования. В листинге 12.2 показана спецификация интерфейса на языке Java для метеостанции. По мере усложнения интерфейсов такой подход оказывается более эффективным, так как для обнаружения ошибок и противоречий в описании интерфейса можно воспользоваться средствами проверки синтаксиса, имеющимися в компиляторе языка программирования. Из представленного описания видно, что некоторые методы могут использовать разное количество параметров. Например, метод **завершение** без параметров применяется к целой станции, а тот же метод с параметрами может отключить один прибор.

Листинг 12.2. Описание интерфейса метеостанции

```
interface Метеостанция {

    public void Метеостанция();

    public void запуск();
    public void запуск (Прибор i);

    public void завершение();
    public void завершение(Прибор i);

    public void отчетПогода();

    public void тестировать();
    public void тестировать(Прибор i);

    public void калибровать(Прибор i);

    public int получитьИдНомер();

    }//Метеостанция
```

12.3. Модификация системной архитектуры

Главное преимущество объектно-ориентированного подхода к проектированию системы состоит в том, что он упрощает задачу внесения изменений в системную архитектуру, поскольку представление состояния объекта не оказывает на нее никакого влияния. Изменение внутренних данных объекта не должно влиять на другие объекты системы. Более того, так как объекты слабо связаны между собой, обычно новые объекты просто вставляются без значительных воздействий на остальные компоненты системы.

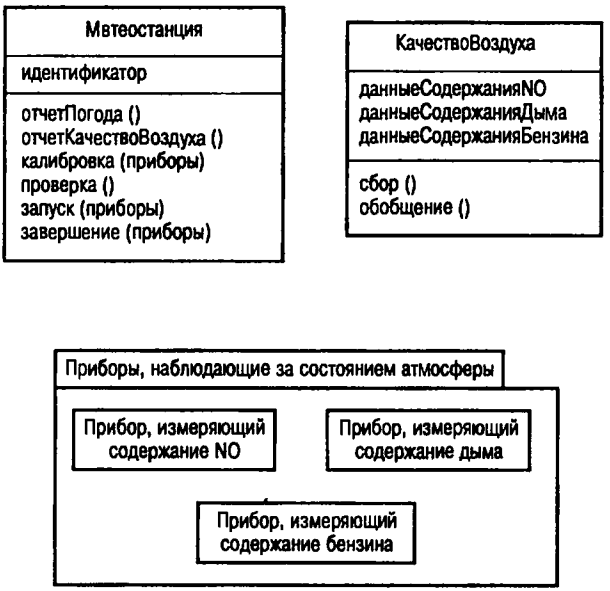


Рис. 12.13. Новые объекты для наблюдения за загрязнением воздуха

Чтобы проиллюстрировать стабильность объектно-ориентированного подхода, предположим, что в каждую метеостанцию потребовалось добавить возможность наблюдения за степенью загрязнения окружающей среды, т.е. необходимо добавить приборы, измеряющие состав воздуха, чтобы вычислить количество различных загрязнителей. Снятые измерения по загрязнению воздуха передаются с таким же интервалом времени, что и остальные метеорологические данные. Для модификации проекта необходимо внести ряд изменений.

1. Класс объектов, именуемый КачествоВоздуха следует вставить как часть объекта Метеостанция на одном уровне с объектом МетеоДанные.
2. В объект Метеостанция необходимо добавить метод отчетКачествоВоздуха, чтобы информация о состоянии воздуха отправлялась на центральный компьютер. Программу управления метеостанцией необходимо изменить так, чтобы при получении запроса с верхнего уровня объекта Метеостанция осуществлялся автоматический сбор данных по загрязнению воздуха.
3. Необходимо добавить объекты, которые представляют типы приборов измеряющих степень загрязнения воздуха. В нашем примере можно добавить приборы, которые измеряли бы уровень оксида натрия, дыма и паров бензина.

На рис. 12.13 показан объект Метеостанция и новые объекты, добавленные в систему. За исключением самого верхнего уровня системы (объект Метеостанция) в имеющиеся объекты не потребовалось вносить изменений. Добавление в систему сбора данных о загрязнении воздуха не оказало никакого влияния на сбор метеорологических данных.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- При объектно-ориентированном проектировании основные компоненты программной системы представляются как объекты со своими состояниями и операциями.
- Объекты предоставляют сервисы (методы) другим объектам и создаются в реальном времени на основе определения класса объектов.
- Объекты могут быть реализованы последовательно и параллельно. Параллельный объект может быть пассивным, у которого состояние изменяется только через его интерфейс, или активным, который может изменять свое состояние без вмешательства извне.
- Унифицированный язык моделирования UML создан для поддержания систем нотаций, которые применяются при документировании объектно-ориентированных проектов.
- Процесс объектно-ориентированного проектирования состоит из следующих этапов: проектирование архитектуры системы, идентификация объектов системы, описание архитектуры различными моделями объектов и документирование интерфейсов объектов.
- В процессе объектно-ориентированного проектирования возможно создание ряда различных моделей. Все модели можно разделить на статические (модели классов, модели обобщения, модели агрегирования) и динамические (модели последовательностей, модели конечного автомата).
- Следует четко определять интерфейсы объектов, так как они используются другими объектами. Для документирования интерфейсов объектов можно использовать языки программирования, например Java.
- Важным преимуществом объектно-ориентированного проектирования является то, что он упрощает процесс модификации системы.

Упражнения

12.1. Объясните, почему в проектировании систем применение подхода, который полагается на слабо связанные объекты, скрывающие информацию о своем представлении, приводит к созданию системной архитектуры, которую затем можно легко модифицировать.

12.2. Покажите на примерах разницу между объектом и классом объектов.

12.3. При каких условиях можно разрабатывать систему, в которой объекты выполняются параллельно?

12.4. С помощью графической системы нотации UML спроектируйте следующие классы объектов с определенными атрибутами и операциями:

- телефон;
- принтер персонального компьютера;
- персональная стереосистема;
- банковские расчеты;
- каталог библиотеки.

12.5. Разработайте более детальный проект метеостанции, добавив описания интерфейсов объектов, изображенных на рис. 12.9. Они могут быть записаны с помощью языков Java, C++ или UML.

12.6. Разработайте проект метеостанции, показывающий взаимодействие между подсистемой сбора данных и приборами, собирающими данные. Воспользуйтесь диаграммой последовательностей.

12.7. Определите возможные объекты в следующих системах, применяя при этом объектно-ориентированный подход.

- Система "Дневник группы" поддерживает расписание собраний и встреч в группе сотрудников. Для организации встречи, в которой участвует группа людей, система находит общие для всех личных дневников свободные "окна" и назначает эту встречу на определенное время. Если система не находит общих "окон", то начинает взаимодействовать с пользователями, чтобы реорганизовать личные дневники и тем самым создать "окно" для встречи.
- Установлена полностью автоматизированная бензоколонка. Водитель вставляет кредитную карточку в считывающее устройство, связанное с насосом; карточка по линиям коммуникаций проверяется кредитной компанией, устанавливается требуемое количество бензина. Затем автомобиль заправляется горючим. Когда подача прекращается, с кредитной карточки водителя снимается стоимость полученного бензина. Кредитная карточка возвращается после вычета водителю. Если карточка неверна, она возвращается водителю перед подачей топлива.

12.8. Запишите точные определения интерфейсов на языке Java или C++ для объектов, определенных в упражнении 12.7.

12.9. Нарисуйте диаграмму последовательностей, в которой отображены взаимодействия между объектами в системе "Дневник группы".

12.10. Нарисуйте диаграмму состояний, на которой отображены возможные изменения состояний для одного или более объектов, определенных в упражнении 12.7.

13

Проектирование систем реального времени

Цели

Цель настоящей главы — познакомить с технологией проектирования систем реального времени и с некоторыми общими архитектурами таких систем. Прочитав эту главу, вы должны:

- знать основные концепции систем реального времени и понимать, почему эти системы обычно реализованы в виде параллельных процессов;
- освоить основные этапы процесса проектирования систем реального времени;
- знать назначение управляющей программы системы реального времени;
- познакомиться с общими архитектурами процессов систем наблюдения и управления, а также систем сбора данных.

Содержание

В настоящее время компьютеры применяются для управления широким спектром разнообразных систем, начиная от простых домашних устройств и заканчивая крупными промышленными комплексами. Эти компьютеры непосредственно взаимодействуют с аппаратными устройствами. Программное обеспечение таких систем (управляющий компьютер плюс управляемые объекты) представляет собой встроенную систему реального времени, задача которой – реагировать на события, генерируемые оборудованием, т.е. в ответ на эти события вырабатывать управляющие сигналы. Такое ПО *встраивается* в большие аппаратные системы и должно обеспечивать реакцию на события, происходящие в окружении системы, в режиме *реального времени*.

Системы реального времени отличаются от других типов программных систем. Их корректное функционирование зависит от способности системы реагировать на события через заданный (как правило, короткий) интервал времени. Вот как я определяю систему реального времени.

> Система реального времени – это программная система, правильное функционирование которой зависит от результатов ее работы и от периода времени, в течение которого получен результат. "Мягкая" система реального времени – это система, в которой операции *удаляются*, если в течение определенного интервала времени не выдан результат. "Жесткая" система реального времени – это система, операции которой становятся *некорректными*, т.е. вырабатывается сигнал об ошибке, если в течение определенного интервала времени результат не выдан.

Систему реального времени можно рассматривать как систему "стимул–отклик". При получении определенного входного стимула (входного сигнала) система генерирует связанный с ним отклик (ответное действие или ответный сигнал). Следовательно, поведение системы реального времени можно определить с помощью списка входных сигналов, получаемых системой, связаниых с ними ответных сигналов (откликов) и интервала времени, в течение которого система должна отреагировать на входной сигнал.

Входные сигналы делятся на два класса.

1. *Периодические сигналы* происходят через предопределенные интервалы времени. Например, система проверяет датчик каждые 50 миллисекунд, и предпринимает действия (реагирует) в зависимости от значений, полученных от датчика (стимула).
2. *Апериодические сигналы* происходят нерегулярно. Обычно они "сообщают о себе" посредством механизма прерываний. Примером апериодического сигнала может быть прерывание, которое вырабатывается по завершении передачи вход/выход и размещения данных в буфере обмена.

В системах реального времени периодические входные сигналы обычно генерируются сенсорами (датчиками), взаимодействующими с системой. Они предоставляют информацию о состоянии внешнего окружения системы. Системные отклики (ответные сигналы) направляются группе исполнительных механизмов, управляющих аппаратными устройствами, которые затем воздействуют на окружение системы. Апериодические входные сигналы могут генерироваться и сенсорами, и исполнительными механизмами. Как правило, апериодические сигналы означают исключительные ситуации, например ошибки в работе аппаратуры. На рис. 13.1 показана модель "сенсор–система–исполнительный механизм" для встроенной системы реального времени.

Рис. 13.1. Общая модель системы реального времени

Системы реального времени должны реагировать на входные сигналы, происходящие в разные моменты времени. Следовательно, архитектуру такой системы необходимо организовать так, чтобы управление переходило к соответствующему обработчику как можно быстрее после получения входного сигнала. В последовательных программах такой механизм передачи управления невозможен. Поэтому обычно системы реального времени проектируют как множество параллельных взаимодействующих процессов. Часть системы – управляющая программа, часто называемая диспетчером, – управляет всеми процессами.

Большинство моделей "стимул-отклик" систем реального времени сводятся к обобщенной архитектурной модели, состоящей из трех типов процессов (рис. 13.2). Для каждого типа сенсора имеется процесс управления сенсором; вычислительный процесс определяет необходимый ответный сигнал на полученный системой входной сигнал; процессы управления исполнительными механизмами управляют действиями этих механизмов. Такая модель позволяет быстро собрать данные со всех имеющихся сенсоров (до того, как произойдет следующий ввод данных), обработать их и получить ответный сигнал от соответствующего исполнительного механизма.

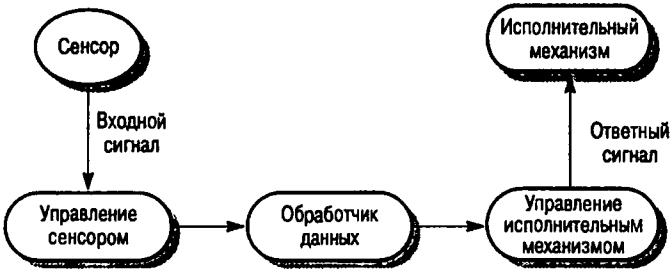


Рис. 13.2. Процессы управления сенсорами и исполнительными механизмами

13.1. Проектирование систем

Как указывалось в главе 2, в процессе проектирования системы принимаются решения о том, какие свойства будут реализованы программной частью системы, а какие – аппаратными средствами. Временные ограничения и другие требования предполагают, что

некоторые функции системы, например обработку сигналов, необходимо реализовать на специально разработанном оборудовании. Таким образом, процесс проектирования систем реального времени включает в себя проектирование оборудования (аппаратуры) специального назначения и проектирование программного обеспечения.

Аппаратные компоненты обеспечивают более высокую производительность, чем эквивалентное им (по выполняемым функциям) программное обеспечение. Аппаратным средствам можно поручить "узкие" места системной обработки сигналов и, таким образом, избежать дорогостоящей оптимизации ПО. Если за производительность системы отвечают аппаратные компоненты, при проектировании ПО основное внимание можно уделить его переносимости, а вопросы, связанные с производительностью, отходят на второй план.

Решения о распределении функций по аппаратным и программным компонентам следует принимать как можно позже, поскольку архитектура системы должна состоять из автономных компонентов, которые можно реализовать как аппаратно, так и программно. Именно такая структура соответствует целям разработчика, проектирующего удобную в обслуживании систему. Следовательно, результатом процесса проектирования высококачественной системы должна быть система, которую можно реализовать и аппаратными и программными средствами.

Отличие процесса проектирования систем реального времени от других систем состоит в том, что уже на первых этапах проектирования необходимо учитывать время реакции системы. В центре процесса проектирования системы реального времени — события (входные сигналы), а не объекты или функции. Процесс проектирования таких систем состоит из нескольких этапов.

1. Определение множества входных сигналов, которые будут обрабатываться системой, и соответствующих им системных реакций, т.е. ответных сигналов.
2. Для каждого входного сигнала и соответствующего ему ответного сигнала вычисляются временные ограничения. Они применяются к обработке как входных, так и ответных сигналов.
3. Объединение процессов обработки входных и ответных сигналов в виде совокупности параллельных процессов. В корректной модели системной архитектуры каждый процесс связан с определенным классом входных и ответных сигналов (как показано на рис. 13.2).
4. Разработка алгоритмов, выполняющих необходимые вычисления для всех входных и ответных сигналов. Чтобы получить представление об объемах вычислительных и временных затрат в процессе обработки сигналов, разработка алгоритмов обычно проводится на ранних этапах процесса проектирования.
5. Разработка временного графика работы системы.
6. Сборка системы, работающей под управлением диспетчера — управляющей программы.

Конечно, описанный процесс проектирования является итерационным. Как только определена структура вычислительных процессов и временной график работы, необходимо сделать всесторонний анализ и провести имитацию работы системы, чтобы удостовериться в том, что она удовлетворяет временным ограничениям. В результате анализа может обнаружиться, что система не отвечает временным требованиям. В таком случае для повышения производительности системы необходимо изменить структуру вычислительных процессов, алгоритм управления, управляющую программу или все эти компоненты вместе.

В системах реального времени сложно анализировать временные зависимости. Из-за непредсказуемой природы апериодических входных сигналов разработчики вынуждены делать некоторые предварительные предположения относительно вероятности появлений апериодических сигналов. Сделанные предположения могут оказаться неверными, и после разработки системы ее показатели производительности не будут удовлетворять временным требованиям. В работах [86, 62] обсуждаются общие проблемы проверки временных параметров систем. В книге [132] всесторонне рассмотрены методы, используемые при анализе производительности систем реального времени.

Все процессы в системе реального времени должны быть скоординированы. Механизм координации процессов обеспечивает исключение конфликтов при использовании общих ресурсов. Когда один процесс использует общий ресурс (или объект), другие процессы не должны иметь доступ к этому ресурсу. К механизмам, обеспечивающим взаимное исключение процессов, относятся семафоры [97], мониторы [162] и метод критических областей [59]. Здесь я не буду рассматривать эти механизмы, так как все они хорошо документированы в описаниях операционных систем [332, 318].

13.1.1. Моделирование систем реального времени

Системы реального времени должны реагировать на события, происходящие через нерегулярные интервалы времени. Такие события (или входные сигналы) часто приводят к переходу системы из одного состояния в другое. Поэтому одним из способов описания систем реального времени может быть модель конечного автомата и соответствующая диаграмма состояний, рассмотренные в главе 7.

В модели конечного автомата в каждый момент времени система находится в одном из своих состояний. Получив входной сигнал, она переходит в другое состояние. Например, система управления клапаном может перейти из состояния "Клапан открыт" в состояние "Клапан закрыт" после получения определенной команды оператора (входной сигнал).

Описанный выше подход к моделированию системы я проиллюстрирую на рассмотренном в главе 7 примере микроволновой печи. На рис. 13.3 показана модель конечного автомата для обычной микроволновой печи, оборудованной кнопками включения питания, таймера и запуска системы. Состояния системы обозначены скругленными прямоугольниками, входные сигналы, вызывающие переход системы из одного состояния в другое, показаны стрелками. На диаграмме показаны все состояния печи, также названы действия исполнительных механизмов системы или действия по выводу информации.

Просматривать последовательность работы системы нужно слева направо. В начальном состоянии **Ожидание**, пользователь может выбрать режим полной или половинной мощности. Следующее состояние наступает при нажатии на кнопку таймера и установке времени работы печи. Если дверь печи закрыта, система переходит в состояние **Действие**. В этом состоянии идет процесс приготовления пищи, после завершения которого печь возвращается в состояние **Ожидание**.

Модели конечного автомата — хороший способ представления структуры систем реального времени. Поэтому такие модели являются неотъемлемой частью методов проектирования систем реального времени [338]. Метод Харела (Harel) [115], базирующийся на *диаграммах состояний*, направлен на решение проблемы внутренней сложности моделей конечного автомата. Диаграмма состояний структурирует модели таким образом, что группы состояния можно было бы рассматривать как единые сущности. Кроме того, с помощью диаграмм состояний параллельные системы можно представить в виде модели состояний. Модели состояний поддерживаются также UML [304, 30*]. В этой книге я также использую систему нотации, принятую в UML.

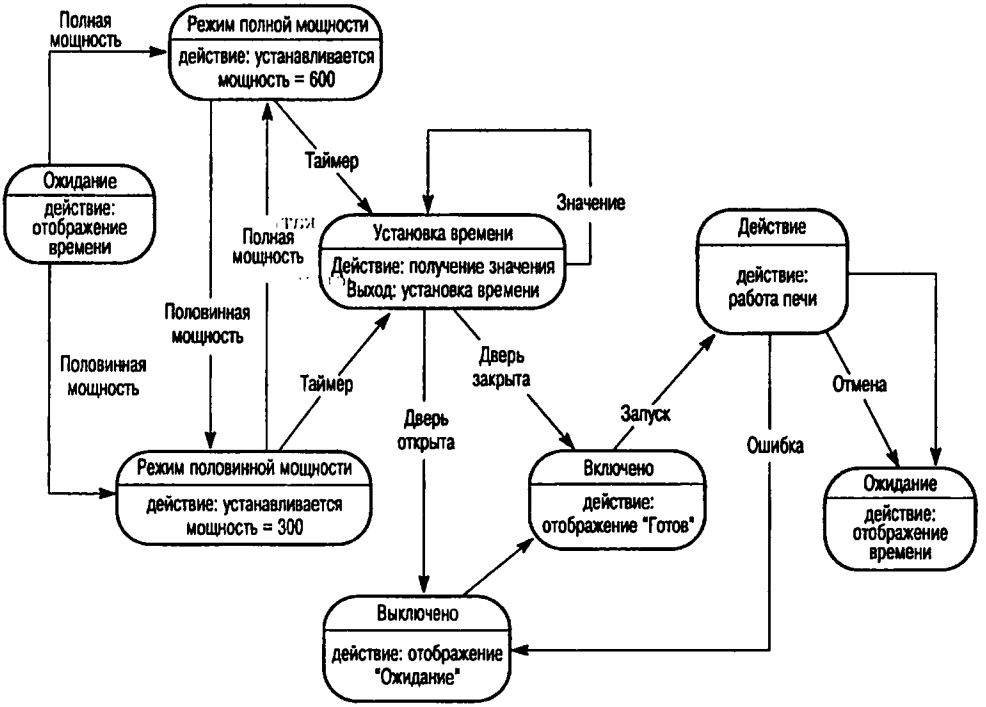


Рис. 13.3. Модель конечного автомата для микроволновой печи

13.1.2. Программирование систем реального времени

На архитектуру системы реального времени оказывает влияние язык программирования, который используется для реализации системы. До сих пор жесткие системы реального времени часто программируются на ассемблерных языках. Языки более высокого уровня также дают возможность сгенерировать эффективный программный код. Например, язык C позволяет писать весьма эффективные программы. Однако в нем нет конструкций, поддерживающих параллельность процессов и управление совместно используемыми ресурсами. Кроме того, программы на C часто сложны для понимания.

Язык Ada изначально разрабатывался для реализации встроенных систем, а потому располагает такими средствами, как управление процессами, исключения и правила представления. Его средство *рандеву* (rendezvous) — отличный механизм для синхронизации задач (процессов) [63, 24, 4*]. К сожалению, первая версия языка Ada (Ada 83) оказалась непригодной для реализации жестких систем реального времени. В ней отсутствовали средства, позволяющие установить предельные сроки завершения задач, не было встроенных исключений для случая превышения предельных сроков и предлагался строгий алгоритм обслуживания очереди "первым пришел — первым вышел". При пересмотре стандартов языка Ada [23] главное внимание уделялось именно этим моментам. В пересмотренной версии языка поддерживаются защищенные типы, что позволило более просто реализовывать защищенные разделяемые структуры данных и обеспечивать более полный контроль при выполнении и синхронизации задач. Однако при программировании систем

реального времени улучшенная версия языка Ada все-таки не обеспечивает достаточного контроля над жесткими системами реального времени.

Первые версии языка Java разрабатывались для создания небольших встраиваемых систем, таких, например, как контроллеры устройств и приборов. Разработчики Java включили несколько средств для поддержки параллельных процессов в виде параллельных объектов (потоков) и синхронизированных методов. Но поскольку в подобных системах нет строгих временных ограничений, то и в языке Java не предусмотрены средства, позволяющие управлять планированием потоков или запускать потоки в конкретные моменты времени.

Поэтому Java не подходит для программирования жестких систем реального времени или систем, в которых имеется строгий временной график процессов. Перечислим основные проблемы Java как языка программирования систем реального времени.

1. Нельзя указать время, в течение которого должен выполняться поток.
2. Неконтролируем процесс очистки памяти — он может начаться в любое время. Поэтому невозможно предсказать поведение потоков во времени.
3. Нельзя определить размеры очереди, связанной с разделяемыми ресурсами.
4. Реализация виртуальной машины Java отличается для разных компьютеров.
5. В языке нет средств для детального анализа распределения времени работы процессоров.

В настоящее время ведется работа по решению некоторых из этих проблем и формируется новая версия языка Java для программирования систем реального времени [256]. Однако не совсем понятно, каким образом эту версию можно отделить от лежащей в ее основе виртуальной машины Java: свойство переносимости языка всегда конфликтовало с характеристиками режима реального времени.

13.2. Управляющие программы

Управляющая программа (диспетчер) системы реального времени является аналогом операционной системы компьютера. Она управляет процессами и распределением ресурсов в системах реального времени, запускает и останавливает соответствующие процессы для обработки входных сигналов и распределяет ресурсы памяти и процессора. Однако обычно в управляющих программах отсутствуют более сложные средства, присущие операционным системам, например средства управления файлами.

В работе [17] представлен полный обзор средств, необходимых управляющим программам систем реального времени. Данная тема обсуждается в монографии [80], где также кратко рассмотрены коммерческие разработки управляющих программ для систем реального времени. Несмотря на то что на рынке программных продуктов существует несколько управляющих программ систем реального времени, их часто проектируют самостоятельно как части систем из-за специальных требований, предъявляемых к конкретным системам реального времени.

Компоненты управляющей программы (рис. 13.4) зависят от размеров и сложности проектируемой системы реального времени. Обычно управляющие программы, за исключением самых простых, состоят из следующих компонентов.

1. *Часы реального времени* периодически предоставляют информацию для планирования процессов.
2. *Обработчик прерываний* управляет апериодическими запросами к сервисам.

3. *Планировщик* просматривает список процессов, которые назначены на выполнение, и выбирает один из них.
4. *Администратор ресурсов*, получив процесс, запланированный на выполнение, выделяет необходимые ресурсы памяти и процессора.
5. *Диспетчер* запускает на выполнение какой-либо процесс.

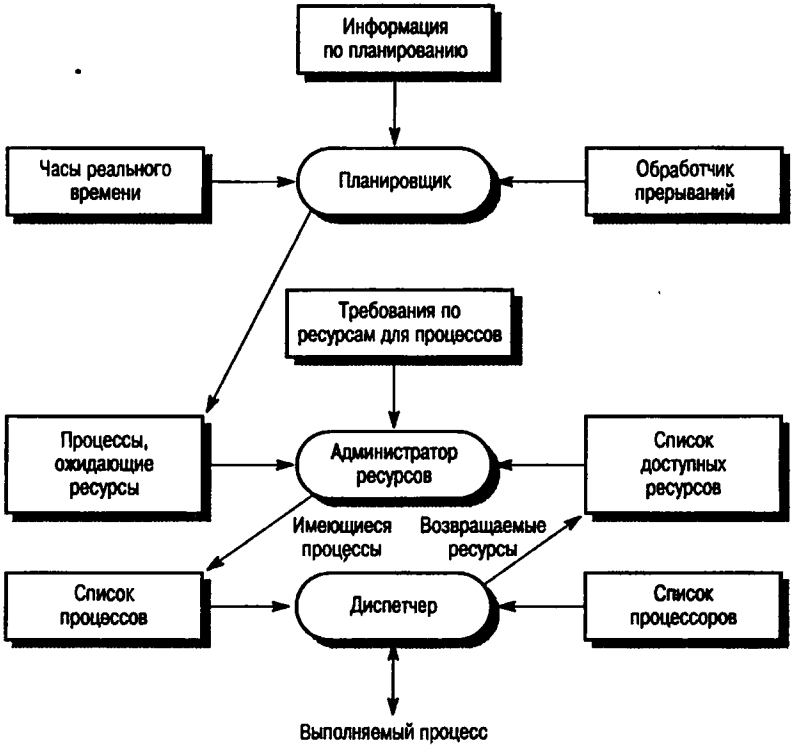


Рис. 13.4. Компоненты управляющей программы реального времени

Управляющие программы систем, предоставляющих сервисы на постоянной основе, например телекоммуникационных или мониторинговых систем с высокими требованиями к надежности, могут иметь еще несколько компонентов.

- *Конфигуратор* отвечает за динамическое переконфигурирование аппаратных средств [205]. Не прекращая работу системы, из нее можно извлечь аппаратные модули и изменить систему посредством добавления новых аппаратных средств.
- *Менеджер неисправностей* отвечает за обнаружение аппаратных и программных неисправностей и предпринимает соответствующие действия по их исправлению. Вопросы отказоустойчивости и восстановления систем рассматриваются в главе 18.

Входные сигналы, обрабатываемые системой реального времени, обычно имеют несколько уровней приоритетов. Для одних сигналов, например связанных с исключительными ситуациями, важно, чтобы их обработка завершалась в течение определенного интервала времени. Если процесс с более высоким приоритетом запрашивает сервис, то выполнение других процессов должно быть приостановлено. Вследствие этого адми-

нистратор системы должен уметь управлять по крайней мере двумя уровнями приоритетов системных процессов.

1. *Уровень прерываний* является наивысшим уровнем приоритетов. Он присваивается тем процессам, на которые необходимо быстро отреагировать. Примером такого процесса может быть процесс часов реального времени.
2. *Тактовый уровень* приоритетов присваивается периодическим процессам.

Еще один уровень приоритетов может быть у фоновых процессов, на выполнение которых не накладываются жесткие временные ограничения, (например, процесс самотестирования). Эти процессы выполняются тогда, когда есть свободные ресурсы процессора.

Внутри каждого уровня приоритетов разным классам процессов можно назначить другие приоритеты. Например, может быть несколько уровней прерываний. Во избежание потери данных прерывание от более быстрого устройства должно вытеснять обработку прерываний от более медленного устройства.

13.2.1. Управление процессами

Управление процессами – это выбор процесса на выполнение, выделение для него ресурсов памяти и процессора и запуск процесса.

Периодическими называются процессы, которые должны выполняться через фиксированный предопределенный промежуток времени (например, при сборе данных или управлении исполнительными механизмами). Управляющая программа системы реального времени для определения момента запуска процесса использует свои часы реального времени. В большинстве систем реального времени есть несколько классов периодических процессов с разными периодами (интервалами времени между выполнением процессов) и длительностью выполнения. Управляющая программа должна быть способна в любой момент времени выбрать процесс, назначенный на выполнение.

Часы реального времени конфигурируются так, чтобы периодически подавать тактовый сигнал, период между сигналами составляет обычно несколько миллисекунд. Сигнал часов инициирует процесс на уровне прерываний, который запускает планировщик процессов для управления периодическими процессами. Процесс на уровне прерываний обычно сам не управляет периодическими процессами, поскольку обработка прерываний должна завершаться как можно быстрее.

Действия, выполняемые управляющей программой при управлении периодическими процессами, показаны на рис. 13.5. Планировщик просматривает список периодических процессов и выбирает из него на выполнение один процесс. Выбор зависит от приоритета процесса, периода процесса, предполагаемой длительности выполнения и конечных сроков завершения процесса. Иногда за один период между тактовыми сигналами часов необходимо выполнить два процесса с разными длительностями выполиения. В такой ситуации один процесс необходимо приостановить на время, соответствующее его длительности.

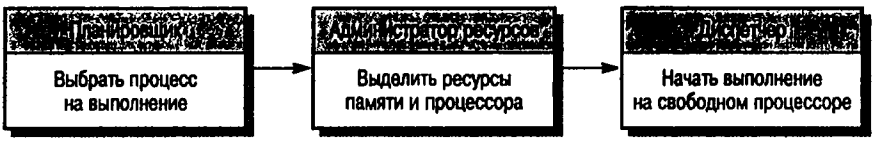


Рис. 13.5. Действия управляющей программы при запуске процесса

Если управляющей программой зарегистрировано прерывание, это означает, что к одному из сервисов сделан запрос. Механизм прерываний передает управление предопре-

деленной ячейке памяти, в которой содержится команда переключения на программу обслуживания прерываний. Эта программа должна быть простой, короткой и быстро выполняться. Во время обслуживания прерываний все другие прерывания системой игнорируются. Чтобы уменьшить вероятность потери данных, время пребывания системы в таком состоянии должно быть минимальным.

Программа, выполняющая сервисную функцию, должна перекрыть доступ следующим прерываниям, чтобы не прервать саму себя. Она должна выявить причину прерывания и инициировать процесс с высоким приоритетом для обработки сигнала, вызвавшего прерывание. В некоторых системах высокоскоростного сбора данных обработчик прерываний сохраняет для последующей обработки данные, которые в момент получения прерывания находились в буфере. После обработки прерывания управление вновь переходит к управляющей программе.

В любой момент времени может быть несколько назначенных на выполнение процессов с разными уровнями приоритетов. Планировщик устанавливает порядок выполнения процессов. Эффективное планирование играет важную роль, если необходимо соответствовать требованиям, которые предъявляются к системе реального времени. Существует две основные стратегии планирования процессов.

1. *Невытесняющее планирование.* Один процесс планируется на выполнение, он запускается и выполняется до конца или блокируется по каким-либо причинам, например при ожидании ввода данных. При таком планировании могут возникнуть проблемы, связанные с тем, что в случае нескольких процессов с разными приоритетами процесс с высоким приоритетом должен ждать завершения процесса с низким приоритетом.
2. *Вытесняющее планирование.* Выполнение процесса может быть приостановлено, если к сервису поступили запросы от процессов с более высоким приоритетом. Процесс с более высоким приоритетом имеет преимущество перед процессом с более низким уровнем приоритета, и поэтому ему выделяется процессор.

В рамках этих стратегий разработано множество различных алгоритмов планирования. К ним относится циклическое планирование, при котором каждый процесс выполняется по очереди, и планирование по скорости, когда при первом выполнении получают более высокий приоритет процессы с коротким периодом выполнения [64]. Каждый из алгоритмов планирования имеет определенные преимущества и недостатки, однако здесь мы их рассматривать не будем.

Информация о назначенном на выполнение процессе передается администратору ресурсов. Он выделяет для выбранного процесса необходимую память, а в многопроцессорной системе — еще и процессор. Затем процесс помещается в "список назначений", т.е. в список процессов, назначенных на выполнение. Когда процессор завершает выполнение какого-либо процесса и становится свободным, вызывается диспетчер. Он просматривает имеющийся список, выбирает процесс, который можно выполнять на свободном процессоре, и запускает его на выполнение.

13.3. Системы наблюдения и управления

В настоящее время можно выделить несколько классов стандартных систем реального времени: мониторинговые системы (системы наблюдения), системы сбора данных, системы управления и др. Каждому типу систем соответствует особая структура процессов, поэтому при проектировании системы, как правило, архитектуру создают по одному из

существующих стандартных типов. Таким образом, вместо обсуждения общих проблем проектирования систем реального времени здесь лучше рассмотреть проектирование с помощью обобщенных моделей.

Системы наблюдения и управления — важный класс систем реального времени. Их основным назначением является проверка сенсоров (датчиков), предоставляющих информацию об окружении системы, и выполнение соответствующих действий в зависимости от поступившей от сенсоров информации. Системы наблюдения выполняют действия после регистрации особого значения сенсора. Системы управления непрерывно управляют аппаратными исполнительными механизмами на основании значений, получаемых от сенсоров.

Рассмотрим следующий пример.

Пусть в здании установлена система охранной сигнализации. В системе используется несколько типов сенсоров: датчики движения, установленные в отдельных комнатах; датчики на окнах первого этажа, которые подают сигнал, если разбивается окно; дверные датчики, фиксирующие открывание дверей. Всего в системе 50 датчиков на окнах, 30 на дверях и 200 датчиков движения.

Когда какой-либо датчик фиксирует присутствие постороннего, система автоматически вызывает местную полицию и, используя звуковой синтезатор, сообщает местоположение датчика (номер комнаты), от которого идет сигнал. В комнатах, расположенных возле активного датчика, включается световая сигнализация и звуковой аварийный сигнал. Система сигнализации обычно включается через сеть, но может работать и от батарей. Проблемы с электропитанием регистрируются специальной программой, контролирующей напряжение электросети. Если в сети регистрируется падение напряжения, программа переключает систему сигнализации на резервное питание от батарей.

В этом примере описана "мягкая" система реального времени, так как здесь нет жестких временных требований. В такой системе не нужно регистрировать события, происходящие с высокой скоростью, поэтому опрос датчиков может проводиться два раза в секунду.

Процесс проектирования начинается с описания апериодических входных сигналов, получаемых системой, и связанных с ними реакций системы. Здесь мы ограничимся упрощенным проектом, в котором не учитываются сигналы, порождаемые процедурами самопроверки, и внешние сигналы, генерируемые при тестировании системы или при ее выключении в случае ложной тревоги. В конечном счете система обрабатывает только два типа входных сигналов.

1. *Отключение электропитания* генерируется программой, контролирующей электрическую цепь. В ответ на этот сигнал система переключает сеть на резервное питание посредством подачи сигнала электронному прибору переключения питания.
2. *Сигнал о вторжении* является входным и генерируется одним из датчиков системы. В ответ на него система определяет номер комнаты, в которой находится активный датчик, вызывает полицию, инициируя звуковой синтезатор, и включает звуковой сигнал тревоги и световую сигнализацию здания в месте нарушения.

На следующем шаге процесса проектирования определяются временные ограничения для каждого входного и ответного сигналов системы. В табл. 13.1 перечислены эти временные ограничения. К разным типам датчиков, генерирующих входные сигналы, предъявлены разные временные требования.

Таблица 13.1. Временные ограничения на входные и ответные сигналы системы

Сигнал	Временные ограничения
Отключение электропитания	Переключение на питание от батарей должно произойти в течение 50 мс
Сигнализация на двери	Каждый сигнальный датчик на дверях проверяется дважды в секунду
Сигнализация на окнах	Каждый датчик на окне проверяется дважды в секунду
Датчик движения	Каждый датчик движения опрашивается дважды в секунду
Звуковой сигнал	Звуковой сигнал должен прозвучать через полсекунды после сигнала датчика
Включение световой сигнализации	Световая сигнализация должна включиться через полсекунды после сигнала датчика
Связь	Вызов в полицию должен начаться в течение 2 с после сигнала датчика
Синтезатор речи	Синтезированное сообщение должно быть готово через 4 с после сигнала датчика

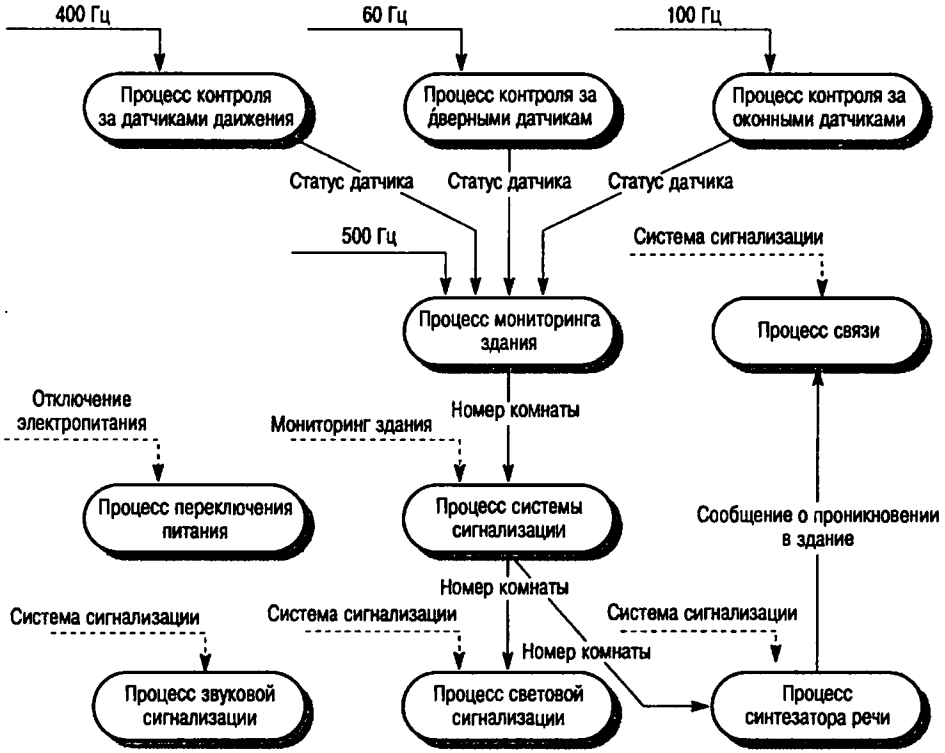


Рис. 13.6. Архитектура процессов системы охранной сигнализации

На следующем этапе проектирования распределяются системные функции по параллельным процессам. Периодически нужно опрашивать три типа датчиков, поэтому у каждого типа датчиков имеется связанный с ними процесс. Кроме этого, есть система управления прерываниями, контролирующая электропитание, система связи с полицией, синтезатор речи, система включения звуковой сигнализации и система, включающая световую сигнализацию возле датчика. Каждой системой управляет независимый процесс. На рис. 13.6 показана архитектура процессов системы.

На схеме, представленной на рис. 13.6, стрелки (с примечаниями), соединяющие отдельные процессы, обозначают потоки данных между процессами с указанием типа данных. Надписи над стрелками справа над каждым процессом указывают систему, управляющую данным процессом. На стрелках вверху указано минимальное значение частоты выполнения процесса.

Частота выполнения процессов определяется количеством датчиков и временными требованиями, предъявляемыми системе. Предположим, что в системе имеется 30 дверных датчиков, которые требуется проверять два раза в секунду. Следовательно, связанный с дверным датчиком процесс должен выполняться 60 раз в секунду (частота 60 Гц). Также 400 раз в секунду выполняется процесс, контролирующий датчик движения.

Апериодические процессы обозначены стрелками с пунктирными линиями. На этих линиях указаны события, которые вызывают данный процесс. Все основные исполнительные процессы (звуковой и световой сигнализации и др.) начинаются командой из процесса Система безопасности; им не нужны данные из других процессов. Процессу, управляющему электропитанием, также не нужны данные из других частей системы.

Все представленные процессы можно реализовать на языке Java как потоки. Листинг 13.1 содержит код Java реализации процесса BuildingMonitor (мониторинг здания), опрашивающего датчики системы. В случае сигнала тревоги программа активизирует систему сигнализации. Пусть в нашем примере система соответствует временным требованиям. Как уже отмечалось, в языке Java 2.0 нет средств для задания частоты выполнения потоков.

Листинг 13.1. Реализация процесса мониторинга здания

```
//См. Web-страницу http://www.software-engin.com/,
//где представлен полный Java-код этого примера
class BuildingMonitor extends Thread {
    BuildingSensor win, door, move;

    Siren siren = new Siren();
    Lights lights = new Lights();
    Synthesizer synthesizer = new Synthesizer();
    DoorSensors doors = new DoorSensors(30);
    WindowSensors windows = new WindowSensors(50);
    MovementSensors movements = new MovementSensors(200);
    PowerMonitor pm = new PowerMonitor();

    BuildingMonitor()
    {
        //инициализация датчиков и запуск процессов
        siren.start();lights.start();
        synthesizer.start();windows.start();
        doors.start();movements.start();pm.start();
    }

    public void run ()
    {
        int room = 0;
        while (true)
        {
```

```
            //проверка датчиков движения два раза в секунду (400 Гц)
            move = movements.getVal();
            //проверка оконных датчиков два раза в секунду (100 Гц)
            win = windows.getVal();
            //проверка дверных датчиков два раза в секунду (60 Гц)
            door = doors.getVal();
            if(move.sensorVal==1|door.sensorVal==1|win.sensorVal==1)
                {
                    //датчик зарегистрировал нарушение
                    if(move.sensorVal == 1) room = move.room;
                    if(door.sensorVal == 1) room = door.room;
                    if(win.sensorVal == 1)  room = win.room;

                    lights.on(room);siren.on();synthesizer.on(room);
                    break;
                }
        }
        lights.shutdown();siren.shutdown();synthesizer.shutdown();
        windows.shutdown();doors.shutdown();movements.shutdown();

    }//run
}//BuildingMonitor
```

Поскольку данная система не содержит строгих временных требований, ее можно реализовать на языке Java. Конечно, в Java 2.0 нет никакой гарантии соответствия временным спецификациям. В нашей системе все датчики опрашиваются одинаковое количество раз, чего не бывает в реальных системах.

После определения архитектуры процессов системы начинается разработка алгоритмов обработки входных сигналов и генерации ответных сигналов. Как уже отмечалось в разделе 3.1, этот этап проектирования необходимо выполнять как можно раньше, чтобы удостовериться, что система будет соответствовать временным требованиям. Если соответствующие алгоритмы оказываются сложными, может возникнуть необходимость в изменении временных ограничений. Обычно алгоритмы систем реального времени сравнительно просты. Они проверяют ячейки памяти, выполняют некоторые простые расчеты и управляют передачей сигнала. В примере системы сигнализации проектирование алгоритмов не рассматривается.

Последним этапом процесса проектирования является составление временного графика выполнения процессов. В нашем примере нет процессов со строгими сроками выполнения. Рассмотрим приоритеты процессов. Все процессы, опрашивающие датчики, имеют один и тот же приоритет. Процессу, управляющему электропитанием, необходимо назначить более высокий приоритет. Приоритеты процессов, управляющих системой сигнализации, и процессов, опрашивающих датчики, должны быть одинаковы.

Систему охранной сигнализации можно отнести скорее к системам наблюдения, чем к системам управления, так как в ней нет исполнительных механизмов, напрямую зависящих от значений датчиков. Примером системы управления может служить система управления отоплением здания. Система наблюдает за температурными датчиками, установленными в разных комнатах здания и переключает нагревательные приборы в зависимости от реальной температуры и температуры, установленной в термореле. Термореле, в свою очередь, контролирует отопительный котел.

Архитектура процессов такой системы показана на рис. 13.7; в общем виде она выглядит подобно системе сигнализации. Дальнейшее рассмотрение этого примера предлагается читателю в качестве упражнения.

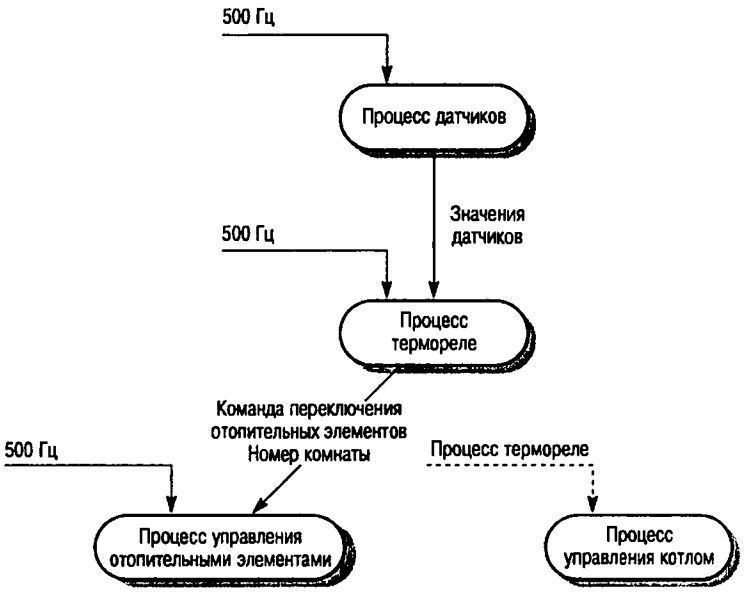


Рис. 13.7. Архитектура процессов системы управления отоплением

13.4. Системы сбора данных

Эти системы представляют другой класс систем реального времени, которые обычно базируются на обобщенной архитектурной модели. Такие системы собирают данные с сенсоров в целях их последующей обработки и анализа.

Для иллюстрации этого класса систем рассмотрим модель, представленную на рис. 13.8. Здесь изображена система, собирающая данные с датчиков, которые измеряют поток нейтронов в ядерном реакторе. Данные, собранные с разных датчиков, помещаются в буфер, из которого затем извлекаются и обрабатываются. На мониторе оператора отображается среднее значение интенсивности потока нейтронов.

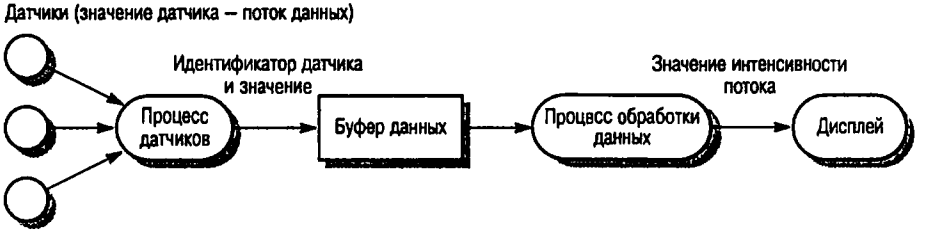


Рис. 13.8. Архитектура системы наблюдения за интенсивностью потока нейтронов

Каждый датчик связан с процессом, который преобразует аналоговый сигнал, показывающий интенсивность входного потока, в цифровой. Сигнал совместно с идентификатором датчика записывается в буфер, где хранятся данные. Процесс, отвечающий за обработку данных, берет их из буфера, обрабатывает и передает процессу отображения для вывода на операторную консоль.

В системах реального времени, ведущих сбор и обработку данных, скорости выполнения и периоды процесса сбора и процесса обработки могут не совпадать. Если обрабатываются большие объемы данных, сбор данных может выполнятся быстрее, чем их обра-

ботка. Если же выполняются только простые вычисления, быстрее происходит обработка данных, а не их сбор.

Чтобы сгладить разницу в скоростях сбора и обработки данных, в большинстве подобных систем для хранения входных данных используется кольцевой буфер. Процессы, создающие данные (процессы-производители), поставляют информацию в буфер. Процессы, обрабатывающие данные (процессы-потребители), берут данные из буфера (рис. 13.9).

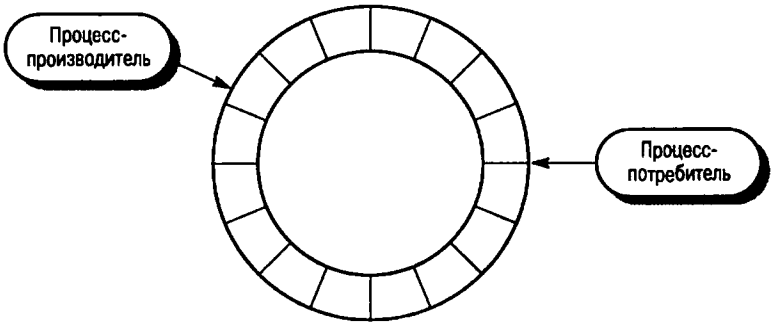


Рис. 13.9. Кольцевой буфер в системе сбора данных

Очевидно, необходимо предотвратить одновременный доступ процесса-производителя и процесса-потребителя к одним и тем же элементам буфера. Кроме того, система должна отслеживать, чтобы процесс-производитель не добавлял данные в полный буфер, а процесс-потребитель не забирал данные из пустого буфера.

В листинге 13.2 показана возможная реализация буфера данных как объекта Java. Значения в буфере имеют тип SensorRecord (запись данных датчика). Определены два метода – get и put: метод get берет элементы из буфера, метод put добавляет элемент в буфер. При объявлении типа CircularBuffer (кольцевой буфер) программный конструктор задает размер буфера.

Листинг 13.2. Реализация кольцевого буфера

```
class CircularBuffer
{
    int bufsize;
    SensorRecord [] store;
    int numberOfEntries = 0;
    int front = 0, back = 0;

    CircularBuffer (int n) {
        bufsize = n;
        store = new SensorRecord [bufsize];
    }//CircularBuffer

    synchronized void put (SensorRecord rec) throws
InterruptedException
    {
        if(numberOfEntries == bufsize)
            wait();
        store [back] = new SensorRecord (rec.sensorId,rec.sensorVal);
        back = back + 1;
        if(back == bufsize)
            back = 0;
        numberOfEntries = numberOfEntries + 1;
        notify();
    }//put
```

```
    synchronized SensorRecord get() throws InterruptedException
    {
        SensorRecord result = new SensorRecord(-1,-1);
        if(numberOfEntries == 0)
            wait();
        result = store [front];
        front = front + 1;
        if(front == bufsize)
            front = 0;
        numberOfEntries = numberOfEntries - 1;
        notify();
        return result;
    }// get
}//CircularBuffer
```

Модификатор synchronized, связанный с методами get и put, указывает на то, что данные методы не должны выполняться параллельно. При вызове одного из этих методов система реального времени блокирует экземпляр объекта, чтобы в это же время не произошел вызов другого метода и соответственно не производились манипуляции на том же участке буфера. Вызовы методов wait и notify из методов get и put гарантируют, что входные данные нельзя положить в полный буфер или взять из пустого буфера. Метод wait вызывает поток и приостанавливается, пока другой поток с помощью метода notify не отправит ему сообщение о снятии ожидания. При вызове метода wait блокировка на защищенные данные объекта снимается. Метод notify возобновляет выполнение одного из ожидающих потоков.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Система реального времени — это программная система, которая должна реагировать на события в реальном масштабе времени. Ее корректное функционирование зависит не только от полученных результатов, но и от времени, в течение которого они получены.
- Общая модель архитектуры систем реального времени состоит из процессов, связанных с каждым классом сенсоров (датчиков) и с каждым исполнительным механизмом. Могут также присутствовать другие координирующие процессы.
- Архитектура системы реального времени обычно организована как множество взаимодействующих между собой параллельных процессов.
- Управляющая программа системы реального времени управляет процессами и аппаратными ресурсами. Обязательным компонентом управляющей программы является планировщик, который запускает процессы на выполнение в заданное время. Планировщик учитывает приоритеты процессов.
- Системы наблюдения и управления периодически опрашивают группу сенсоров, собирающих информацию из окружения системы. Посредством команд и исполнительных механизмов система реагирует на данные, полученные от сенсоров.
- Системы сбора данных обычно организуются в соответствии с моделью "производитель-потребитель". Процесс-производитель помещает данные в кольцевой буфер, где они используются процессом-потребителем. Чтобы исключить конфликты между процессом-производителем и процессом-потребителем, буфер обычно реализуется как процесс.

Упражнения

13.1. Почему системы реального времени обычно реализованы как множество параллельных процессов? Проиллюстрируйте свой ответ примерами.

13.2. Объясните, почему объектно-ориентированные методы разработки ПО не всегда подходят к системам реального времени.

13.3. Нарисуйте диаграммы состояний управляющего ПО для следующих систем.

- Автоматическая стиральная машина с разными программами для разных типов белья.
- Программное обеспечение для проигрывателя компакт-дисков.
- Телефонный автоответчик, который записывает входящие сообщения и отображает количество полученных сообщений на жидкокристаллическом экране. Система должна определить телефон звонившего, вывести на экран последовательность чисел (идентифицированных как тоновый набор) и хранить записанные сообщения, которые затем можно прослушать.
- Автомат по выдаче напитков, который может налить кофе с молоком и сахаром или без них. Пользователь бросает монету и с помощью нажатия кнопок на автомате выбирает нужный режим. Автомат выдает чашку с растворимым кофе. Пользователь затем ставит чашку под кран, нажимает другую кнопку и автомат наливает в чашку горячую воду.

13.4. Используя методы проектирования систем реального времени спроектируйте заново систему сбора данных от метеостанций, рассмотренную в главе 12, в виде системы "стимул-ответ".

13.5. Спроектируйте архитектуру процессов для системы наблюдения, собирающей данные с группы датчиков, измеряющих состав воздуха и расположенных вокруг города. В системе 5000 датчиков, организованных в группы по 100 штук. Каждый датчик должен проверяться 4 раза в секунду. Если более 30% датчиков в группе зафиксируют, что качество воздуха ниже допустимого уровня, активизируется предупреждающий световой сигнал. Все датчики передают собранные данные центральному компьютеру, который каждые 15 мин генерирует отчет о составе воздуха в городе.

13.6. Обсудите сильные и слабые стороны Java как языка программирования для реализации систем реального времени.

13.7. Система безопасности поезда автоматически закрывает двери, если скорость поезда превышает предельную для данного участка трассы или если при выходе на участок пути горит красный свет (т.е. въезд на участок запрещен). Остальные подробности перечислены во врезке 13.1. Идентифицируйте входные сигналы, которые должна обрабатывать бортовая система управления поездом, и связанные с ними ответные сигналы.

13.8. Предположите вероятную архитектуру процессов такой системы.

13.9. Если в бортовой системе безопасности поезда при сборе данных с путевых передатчиков используются периодические процессы, какую частоту сбора данных следует запланировать, чтобы система гарантированно получала информацию от передатчиков? Обоснуйте свой ответ.

Врезка 13.1. Описание системы безопасности поезда

- Система собирает данные о скорости на участке от путевого передатчика, который непрерывно передает идентификатор участка и значение скорости на этом участке. Этот же передатчик передает информацию о статусе сигнала управления на участке трассы. Время передачи всей информации не должно превышать 50 мс.
- Чтобы получить данные от передатчика, расстояние между поездом и передатчиком не должно превышать 10 м.
- Максимальная скорость поезда 180 км/ч.
- Датчики на поезде предоставляют информацию о текущей скорости поезда (обновляемую каждые 250 мс), статус поезда обновляется каждые 100 мс.
- Если на текущем участке скорость поезда превышает предельную более чем на 5 км/ч, в кабине машиниста раздается предупреждающий сигнал. При превышении предельной скорости более чем на 10 км/ч начинается автоматическое торможение поезда, которое продолжается до тех пор, пока скорость поезда не будет равна предельной на данном участке. Торможение поезда должно начинаться через 100 мс после регистрации повышенной скорости поезда.
- Если поезд выехал на участок, перед которым горит красный свет семафора, система безопасности должна затормозить поезд до полной остановки. Торможение должно начаться через 100 мс после регистрации красного светового сигнала от семафора.
- Система постоянно обновляет информацию на экране в кабине машиниста.

14

Проектирование с повторным использованием компонентов

Цели

Цель настоящей главы – показать различные способы повторного использования имеющегося программного обеспечения в процессе проектирования программных систем. Прочитав эту главу, вы должны:

- знать основные преимущества повторного использования компонентов ПО и проблемы, которые могут возникнуть при этом;
- познакомиться с различными типами повторно используемых компонентов и знать основные этапы процесса их проектирования;
- усвоить, что такое семейства приложений и почему они служат эффективным способом повторного использования ПО;
- знать, что паттерны – это абстракции высокого уровня, которые обеспечивают повторное использование компонентов в процессе объектно-ориентированного проектирования.

Содержание

В большинстве инженерных разработок процесс проектирования основан на повторном использовании уже имеющихся компонентов. В таких сферах, как механика или электротехника, инженеры никогда не разрабатывают проект "с нуля". Их проекты базируются на компонентах, уже проверенных и протестированных в других системах. Как правило, это не только малые компоненты, например фланцы и клапаны, но также целые подсистемы, например двигатели, компрессоры или турбины.

В настоящее время не вызывает сомнений тот факт, что необходимо сравнивать различные подходы к разработке программного обеспечения. Если программное обеспечение рассматривать как актив, то повторное использование этих активов позволит существенно сократить расходы на его разработку. Только с помощью систематического повторного использования ПО можно уменьшить расходы на его создание и обслуживание, сократить сроки разработки систем и повысить качество программных продуктов.

Чтобы повторное использование ПО было эффективным, его необходимо учитывать на всех этапах процесса проектирования ПО или процесса разработки требований. Во время программирования возможно повторное использование на этапе подбора компонентов, соответствующих требованиям. Однако для *систематического* повторного использования необходим такой процесс проектирования, в ходе которого постоянно рассматривалась бы возможность повторного использования уже существующих архитектур, где система была бы явно организована из доступных имеющихся компонентов ПО.

Метод проектирования ПО, основанный на повторном использовании, предполагает максимальное использование уже имеющихся программных объектов. Такие объекты могут радикально различаться размерами.

1. *Повторно используемые приложения.* Можно повторно использовать целые приложения либо путем включения их в систему без изменения других подсистем (например, коммерческие готовые продукты, см. раздел 14.1.2), либо с помощью разработки семейств приложений, работающих на разных платформах и адаптированных к требованиям конкретных заказчиков (см. раздел 14.2).
2. *Повторно используемые компоненты.* Можно повторно использовать компоненты приложений — от подсистем до отдельных объектов. Например, система распознавания текста, разработанная как часть системы обработки текстов, может повторно использоваться в системах управления базами данных. Этот вид повторного использования рассматривается в разделе 14.3.
3. *Повторно используемые функции.* Можно повторно использовать программные компоненты, которые реализуют отдельные функции, например математические. Основанный на стандартных библиотеках метод повторного использования применяется в программировании последние 40 лет.

Повторное использование целых приложений практикуется довольно широко; при этом компании, занимающиеся разработкой ПО, адаптируют свои системы для разных платформ и для работы в различных условиях. Также хорошо известно повторное использование функций через стандартные библиотеки, например графические и математические. Интерес к повторному использованию компонентов возник еще в начале 1980-х годов, однако на практике такой подход к разработке систем ПО применяется лишь последние несколько лет.

Очевидным преимуществом повторного использования ПО является снижение общей стоимости проекта, так как в целом требуется специфицировать, спроектировать, реализовать и проверить меньшее количество системных компонентов. Но снижение стоимо-

сти проекта — это только потенциальное преимущество повторного использования. Как видно из табл. 14.1, повторное использование ПО имеет ряд других преимуществ.

Таблица 14.1. Преимущества повторного использования ПО

Преимущество	Описание
Повышение надежности	Компоненты, повторно используемые в других системах, оказываются значительно надежнее новых компонентов. Они протестированы и проверены в разных условиях работы. Ошибки, допущенные при их проектировании и реализации, обнаружены и устранены еще при первом их применении. Поэтому повторное использование компонентов сокращает общее количество ошибок в системе
Уменьшение проектных рисков	Для уже существующих компонентов можно более точно прогнозировать расходы, связанные с их повторным использованием, чем расходы, необходимые на их разработку. Такой прогноз — важный фактор администрирования проекта, так как позволяет уменьшить неточности при предварительной оценке сметы проекта
Эффективное использование специалистов	Часть специалистов, выполняющих одинаковую работу в разных проектах, может заниматься разработкой компонентов для их дальнейшего повторного использования, эффективно применяя накопленные ранее знания
Соблюдение стандартов	Некоторые стандарты, такие как стандарты интерфейса пользователя, можно реализовать в виде набора стандартных компонентов. Например, можно разработать повторно используемые компоненты для реализации различных меню пользовательского интерфейса. Все приложения предоставляют меню пользователям в одном формате. Использование стандартного пользовательского интерфейса повышает надежность систем, так как, работая со знакомым интерфейсом, пользователи совершают меньше ошибок
Ускорение разработки	Часто для успешного продвижения системы на рынке необходимо как можно более раннее ее появление, причем независимо от полной стоимости ее создания. Повторное использование компонентов ускоряет создание систем, так как сокращается время на их разработку и тестирование

Для успешного проектирования и разработки ПО с повторным использованием компонентов должны выполняться три основных условия.

1. Возможность поиска необходимых системных компонентов. В организациях должен быть каталог документированных компонентов, предназначенных для повторного использования, который обеспечивал бы быстрый поиск нужных компонентов.
2. При повторном использовании необходимо удостовериться, что поведение компонентов предсказуемо и надежно. В идеале все компоненты, представленные в каталоге, должны быть сертифицированы, чтобы подтвердить соответствие определенным стандартам качества.

3. На каждый компонент должна быть соответствующая документация, цель которой — помочь разработчику получить нужную информацию о компоненте и адаптировать его к новому приложению. В документации должна содержаться информация о том, где используется данный компонент, и другие вопросы, которые могут возникнуть при повторном использовании компонента.

Успешное использование компонентов в приложениях Visual Basic, Visual C++ и Java продемонстрировало важность повторного использования. Разработка ПО, основанная на повторном использовании компонентов, становится широко распространенным рентабельным подходом к разработке программных продуктов [331, 3*].

Вместе с тем подходу к разработке ПО с повторным использованием компонентов присущ ряд недостатков и проблем (табл. 14.2), которые препятствуют запланированному сокращению расходов на разработку проекта.

Таблица 14.2. Проблемы повторного использования

Проблема	Описание
Повышение стоимости сопровождения системы	Недоступность исходного кода компонента может привести к увеличению расходов на сопровождение системы, так как повторно используемые системные элементы могут со временем оказаться не совместимыми с изменениями, производимыми в системе
Недостаточная инструментальная поддержка	CASE-средства не поддерживают разработку ПО с повторным использованием компонентов. Интегрирование этих средств с системой библиотек компонентов затруднительно или даже невозможно. Если процесс разработки ПО осуществляется с помощью CASE-средств, повторное использование компонентов можно полностью исключить
Синдром "изобретения велосипеда"	Некоторые разработчики ПО предпочитают переписать компоненты, так как полагают, что смогут при этом их усовершенствовать. Кроме того, многие считают, что создание программ "с нуля" перспективнее и "благороднее" повторного использования написанных другими программ
Содержание библиотеки компонентов	Заполнение библиотеки компонентов и ее сопровождение может стоить дорого. В настоящее время еще недостаточно хорошо продуманы методы классификации, каталогизации и извлечения информации о программных компонентах
Поиск и адаптация компонентов	Компоненты ПО нужно найти в библиотеке, изучить и адаптировать к работе в новых условиях, что "не укладывается" в обычный процесс разработки ПО

Из перечисленного выше следует, что повторное использование компонентов должно быть систематическим, плановым и включенным во все организационные программы организации-разработчика. В Японии повторное использование известно много лет [231] и является неотъемлемой частью "японского" метода разработки ПО [85]. Многие компании, например Hewlett-Packard, успешно применяют повторное использование в своих разработках [139]. Опыт этой компании представлен в фундаментальной книге [187].

Альтернативой повторному использованию программных компонентов является применение программных генераторов. Согласно этому подходу информация, необходимая для повторного использования, записывается в систему генератора программ с учетом знаний о той предметной области, где будет эксплуатироваться разрабатываемая система. В данном случае в системной спецификации должно быть точно указано, какие именно компоненты выбраны для повторного использования, а также описаны их интерфейсы и то, как они должны компоноваться. На основе такой информации генерируется система ПО (рис. 14.1).

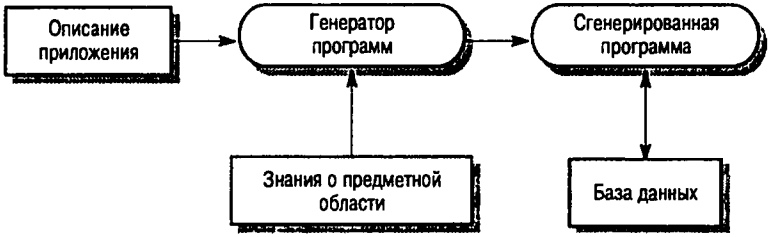


Рис. 14.1. Генерирование программ

Повторное использование, основанное на генераторах программ, возможно только тогда, когда можно идентифицировать предметные абстракции и их отображение в исполняемый код. Поэтому для компоновки и управления предметными абстракциями используются, как правило, проблемно-зависимые языки (например, языки четвертого поколения). Вот предметные области, в которых применение такого подхода может быть успешным.

1. *Генераторы приложений для обработки экономических данных.* На входе генератора — описание приложения на языке четвертого поколения или диалоговая система, где пользователь определяет экранные формы и способы обработки данных. На выходе — программа на каком-либо языке программирования, например COBOL или SQL.
2. *Генераторы программ синтаксического анализатора.* На входе генератора — грамматическое описание языка, на выходе — программа грамматического разбора языковых конструкций.
3. *Генераторы кодов CASE-средств.* На входе генераторов — архитектура ПО, а на выходе — программная реализация проектируемой системы.

Разработка ПО с использованием программных генераторов экономически выгодна, однако существенно зависит от полноты и корректности определения абстракций предметной области. Данный подход можно широко использовать в перечисленных выше предметных областях и в меньшей степени при разработке систем управления и контроля [261]. Главное преимущество этого подхода состоит в относительной легкости разработки программ с помощью генераторов. Однако необходимость глубокого понимания предметной области и ее моделей ограничивает применимость данного метода.

14.1. Покомпонентная разработка

Метод покомпонентной разработки ПО с повторным использованием компонентов появился в конце 1990-х годов как альтернатива объектно-ориентированному подходу

к разработке систем, который не привел к повсеместному повторному использованию программных компонентов, как предполагалось изначально. Отдельные классы объектов оказались слишком детализированными и специфическими: их требовалось связывать с приложением либо во время компиляции, либо при компоновке системы. Использование классов обычно предполагает наличие детальных данных о классах, что делает доступным исходный код, но для коммерческих продуктов исходный код открыт очень редко. Несмотря на ранние оптимистические прогнозы, значительное развитие рынка отдельных программных объектов и компонентов так и не состоялось.

Компоненты более абстрактны, чем классы объектов. Поэтому их можно считать независимыми поставщиками сервисов. Если система запрашивает какой-либо сервис, вызывается компонент, предоставляющий этот сервис независимо от того, где выполняется компонент и на каком языке написан. Примером простейшего компонента может быть отдельная математическая функция, вычисляющая, например, квадратный корень числа. Для вычисления квадратного корня программа вызывает компонент, который может выполнить данное вычисление. На другом конце масштабной линейки компонентов находятся системы, которые предоставляют полный вычислительный сервис.

Взгляд на компонент как на поставщика сервисов определяется двумя основными характеристиками компонентов, допускающими их повторное использование.

1. Компонент — это независимо выполняемый программный объект. Исходный код компонента может быть недоступен, поэтому такой компонент не компилируется совместно с другими компонентами системы.
2. Компоненты объявляют свой интерфейс и все взаимодействия с ними осуществляются с его помощью. Интерфейс компонента описывается в терминах параметризованных операций, а внутреннее состояние компонента всегда скрыто.

Компоненты определяются через свои интерфейсы. В большинстве случаев компоненты можно описать в виде двух взаимосвязанных интерфейсов, как показано на рис. 14.2.

- *Интерфейс поставщика сервисов*, который определяет сервисы, предоставляемые компонентом.
- *Интерфейс запросов*, который определяет, какие сервисы доступны компоненту из системы, использующей этот компонент.

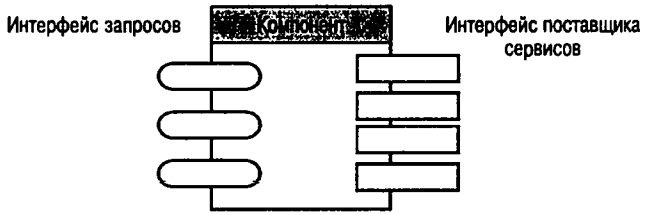


Рис. 14.2. Интерфейсы компонента

В качестве примера рассмотрим компонент (рис. 14.3), который предоставляет сервисы вывода документов на печать. В нашем примере поддерживаются следующие сервисы: печать документов, просмотр состояния очереди на конкретном принтере, регистрация и удаление принтеров из системы, передача документа с одного принтера на другой и удаление документа из очереди на печать. Очень важно, чтобы компьютерная платформа, на

которой исполняется компонент, предоставляла сервис (назовем его ФайлОпПринтер), позволяющий извлечь файл описания принтера, и сервис КомПринтер, передающий команды на конкретный принтер.

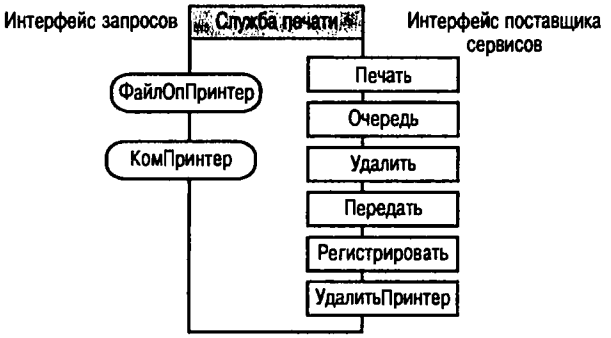


Рис. 14.3. Компонент службы печати

Компоненты могут существовать на разных уровнях абстракции – от простой библиотечной подпрограммы до целых приложений, таких как Microsoft Excel. В работе [236] определено пять уровней абстракции компонентов.

1. *Функциональная абстракция.* Компонент реализует отдельную функцию, например математическую. В сущности, интерфейсом поставщика сервисов здесь является сама функция.
2. *Бессистемная группировка.* В данном случае компонент – это набор слабо связанных между собой программных объектов и подпрограмм, например объявлений данных, функций и т.п. Интерфейс поставщика сервисов состоит из названий всех объектов в группировке.
3. *Абстракции данных.* Компонент является абстракцией данных или классом, описанным на объектно-ориентированном языке. Интерфейс поставщика сервисов состоит из методов (операций), обеспечивающих создание, изменение и получение доступа к абстракции данных.
4. *Абстракции кластеров.* Здесь компонент – это группа связанных классов, работающих совместно. Такие компоненты иногда называют структурой. Интерфейс поставщика сервисов является композицией всех интерфейсов объектов, составляющих структуру (см. раздел 14.1.1).
5. *Системные абстракции.* Компонент является полностью автономной системой. Повторное использование абстракций системного уровня иногда называют повторным использованием коммерческих продуктов. Интерфейсом поставщика сервисов на этом уровне является так называемый программный интерфейс приложений (Application Programming Interface – API), который предоставляет доступ к системным командам и методам. Повторное использование коммерческих продуктов рассматривается в разделе 14.1.2.

Подход покомпонентной разработки систем можно интегрировать в общий процесс создания ПО путем добавления специальных этапов, на которых отбираются и адаптируются повторно используемые компоненты (рис. 14.4). Проектировщики системы разрабатывают системную архитектуру на высоком уровне абстракции, составляя спецификации

системных компонентов. В дальнейшем эти спецификации используются для поиска повторно используемых компонентов. Они включаются в систему либо на уровне системной архитектуры, либо на более низких детализированных уровнях.

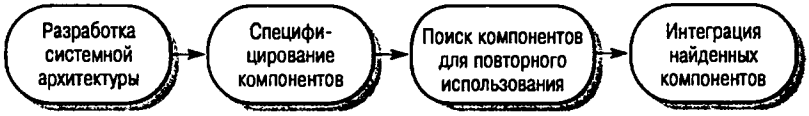


Рис. 14.4. Интеграция повторного использования компонентов в процесс разработки ПО

Хотя такой подход может привести к значительному увеличению количества повторно используемых компонентов, он, в сущности, противоположен подходу, применяемому в других инженерных дисциплинах, где процесс проектирования подчинен идее повторного использования. Перед началом этапа проектирования разработчики выполняют поиск компонентов, подходящих для повторного использования. Системная архитектура строится на основе уже имеющихся (готовых) компонентов (рис. 14.5).

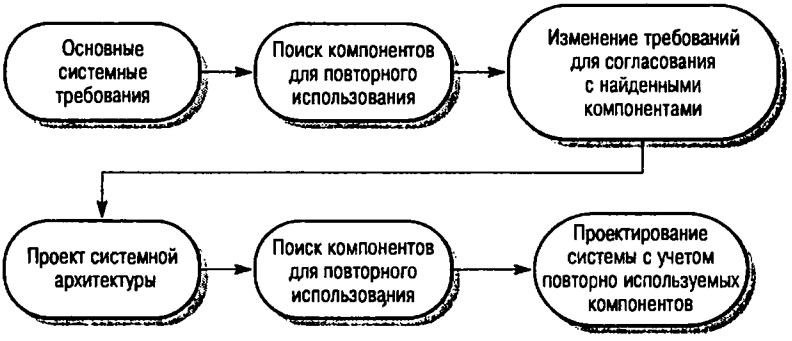


Рис. 14.5. Процесс проектирования с повторным использованием компонентов

В данном случае требования к системе изменяются с учетом имеющихся компонентов, выбранных для повторного использования. Системная архитектура также базируется на имеющихся компонентах. Конечно, такой подход предполагает определенные компромиссы в реализации требований. Хотя такая система может оказаться менее эффективной, чем система, разработанная без повторного использования компонентов, этот недостаток компенсируется более низкой стоимостью разработки, более высокими темпами создания системы и ее повышенной надежностью.

Обычно покомпонентный процесс реализации системы является либо процессом макетирования (прототипирования), либо процессом пошаговой разработки. Вместе с библиотеками компонентов можно использовать стандартные языки программирования, например Java. Альтернативой ему (и более распространенным языком) является язык сценариев, который специально разработан для интегрирования повторно используемых компонентов и обеспечивает быструю разработку программ.

Первым языком сценариев, разработанным для интеграции повторно используемых компонентов, был Unix shell [56]. После него разработано множество других языков сценариев, например Visual Basic и TCL/TK. В работе [268] обсуждаются преимущества языков сценариев, в том числе отсутствие определения типов данных, и тот факт, что они не компилируются, а интерпретируются.

Пожалуй, главной проблемой, связанной с покомпонентной разработкой систем, является их сопровождение и модернизация. При изменении требований к системе часто в компоненты необходимо внести изменения, соответствующие новым требованиям, однако в большинстве случаев это невозможно, поскольку исходный код компонентов недоступен. Также, как правило, не подходит альтернативный вариант: замена одного компонента другим. Таким образом, необходима дополнительная работа по адаптации повторно используемых компонентов, что приводит к повышению стоимости обслуживания системы. Однако, поскольку разработка с повторным использованием компонентов позволяет быстрее создавать ПО, организации согласны оплачивать дополнительные расходы на сопровождение и модернизацию систем.

14.1.1. Объектные структуры приложений

Первые сторонники объектно-ориентированной разработки ПО полагали, что наиболее подходящими абстракциями для повторного использования являются объекты. Однако, как следует из предыдущего раздела, объекты обычно слишком мелкие структурные единицы, чересчур "привязанные" к конкретному приложению. Очевидно, что в процессе объектно-ориентированного проектирования вместо повторного использования объектов намного эффективнее повторно использовать крупномодульные абстракции, так называемые объектные структуры приложения.

Объектные структуры приложения представляют собой структуры подсистем, состоящих из множества абстрактных и конкретных классов объектов и интерфейсов между ними [343]. Отдельные детали подсистем реализуются с помощью компонентов и обеспечивают конкретные реализации абстрактных классов. Объектные структуры, как правило, не являются сами по себе приложениями. Обычно приложения строятся посредством интегрирования нескольких объектных структур.

Существует три основных класса объектных структур [113].

1. *Инфраструктуры систем.* Обеспечивают разработку инфраструктур для систем связи (коммуникационных систем), пользовательских интерфейсов и компиляторов [306].
2. *Интеграционные структуры.* Как правило, состоят из набора стандартов и связанных с ними классов объектов, обеспечивающих взаимодействие и обмен данными между компонентами. К этому типу структур относятся CORBA, COM и DCOM от Microsoft, а также Java Beans [264]. Данный тип объектных структур рассматривался в главе 11 при обсуждении архитектур распределенных объектов.
3. *Структуры инструментальных сред разработки приложений.* Связаны с отдельными прикладными областями, такими как телекоммуникации или финансы [30]. Они встраиваются в систему знаний области приложения и поддерживают разработку приложений конечного пользователя. Данные структуры связаны с семействами приложений, которые рассматриваются в разделе 14.2.

Объектная структура — это обобщенная структура, которую можно детализировать и расширить при создании конкретной подсистемы или приложения. Детализация объектной структуры обычно предполагает добавление конкретных классов, наследующих методы от абстрактных классов объектной структуры. Кроме того, определяются методы, которые вызываются в ответ на события, определенные объектной структурой.

На момент написания книги были достаточно хорошо разработаны инфраструктуры систем, особенно те из них, которые связаны с графическими интерфейсами пользователя. Их постепенно вытесняют структуры инструментальных сред разработки приложений

[77]. Попытаемся разобрать наиболее эффективные представления и организацию данных объектных структур.

Одной из самых известных и распространенных объектных структур для графических интерфейсов пользователя (GUI) является объектная структура "модель–представление–контроллер" (рис. 14.6). Эта модель появилась в 1980-х годах как метод проектирования графических интерфейсов пользователя, который поддерживает различные представления объекта и различает взаимодействия с каждым из этих представлений.

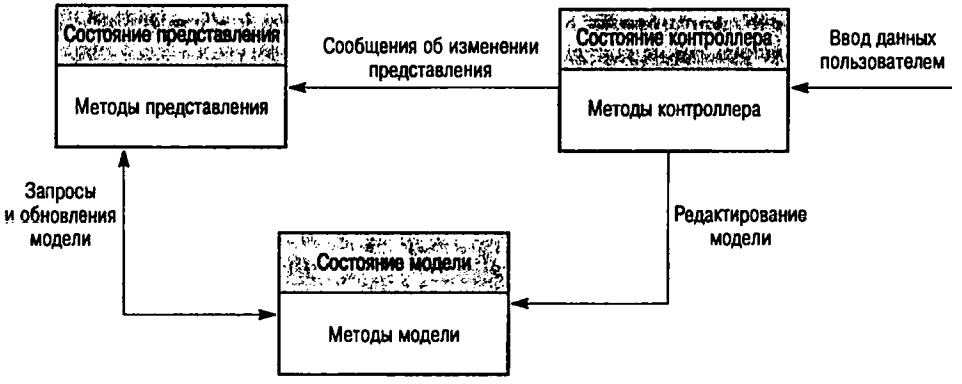


Рис. 14.6. Объектная структура "модель–представление–контроллер"

Объектные структуры часто реализуются в виде паттернов (см. раздел 14.2). Например, объектная структура "модель–представление–контроллер" включена в паттерн Обозреватель, описанный во врезке 14.1, а также в ряд других паттернов, подробно рассмотренных в работе [124].

Основные недостатки объектных структур – их сложность и время, необходимое для того, чтобы научиться работать с ними. Для полного изучения объектных структур может понадобиться несколько месяцев. Именно поэтому в больших организациях некоторые разработчики ПО специализируются по объектным структурам. Нет никаких сомнений в эффективности данного подхода к повторному использованию, однако высокие затраты на изучение объектных структур ограничивают его повсеместное распространение.

14.1.2. Повторное использование коммерческих программных продуктов

Термин "коммерческие программные продукты" можно применить к любому компоненту, созданному независимым производителем. Вместе с тем под этим термином часто подразумевается программное обеспечение системного уровня. Я также предпочитаю говорить о коммерческих системах. Функциональность, предлагаемая этими системами, намного шире функциональности более специализированных компонентов, и поэтому увеличивается потенциальный выигрыш, полученный от повторного использования.

Некоторые типы коммерческих систем используются повторно на протяжении многих лет. Лучшим примером тому, по-видимому, служат базы данных. Очень немногие разработчики создают собственные системы управления базами данных. Но до недавнего времени существовало лишь несколько больших коммерческих систем, таких как системы управления базами данных и мониторы телеобработки, используемые повторно.

Новые подходы к проектированию систем, предоставляющие программам доступ к системным функциям, показывают, что создание больших систем (например, систем элек-

тронной коммерции) посредством интегрирования ряда коммерческих систем сегодня рассматривается как один из приемлемых вариантов проектирования. Благодаря функциональности, предлагаемой этими системами, сокращение финансовых и временных затрат может достичь величины, сравнимой с разработкой нового ПО "с нуля". Более того, уменьшаются риски, так как коммерческие системы уже существуют и разработчики могут увидеть, удовлетворяют ли они предъявляемым к ним требованиям.

В принципе использование крупномодульных коммерческих систем не отличается от использования любого другого более специализированного компонента. Для этого необходимо изучить интерфейсы системы и использовать их для организации взаимодействия с другими системными компонентами, также необходимо разработать системную архитектуру, которая поддерживала бы коммерческие системы при совместной работе.

Однако тот факт, что коммерческие программные продукты представляют собой крупные системы и часто продаются как отдельные автономные системы, виосит дополнительные проблемы. При интеграции таких систем могут возникнуть, как минимум, четыре проблемы [42].

1. *Недостаточный контроль над функциональностью и производительностью коммерческих продуктов.* Хотя считается, что их интерфейсы известны, не исключена вероятность наличия скрытых операций, которые будут "пересекаться" с системными операциями. Решение этой проблемы может стать приоритетом для системных разработчиков, использующих коммерческие продукты, причем эта проблема, очевидно, не связана с производителем данного продукта.
2. *Проблемы, связанные с организацией взаимодействия коммерческих систем.* Иногда сложно подобрать коммерческие продукты для совместной работы, поскольку каждый продукт разрабатывается на основе различных предположений по поводу его использования. В [127] приведены результаты эксперимента по интегрированию четырех различных коммерческих продуктов, при этом было установлено, что работа трех продуктов основывалась на одних и тех же событиях, однако все они использовали разные модели событий и каждый из них считал, что имеет первоочередной доступ к очереди событий. Как следствие, на разработку системы было потрачено в пять раз больше усилий, чем предполагалось, а на составление временного графика работы системы ушло два года вместо предполагавшихся шести месяцев.
3. *Отсутствие контроля за модификацией коммерческих продуктов.* Производители коммерческих продуктов принимают решения по изменению своих систем под давлением рынка. В частности, новые версии программных продуктов, разработанных для персональных компьютеров, создаются очень часто и могут оказаться не совместимыми с предыдущими версиями. Новые версии могут обладать дополнительной функциональностью, неподдерживаемой предыдущими версиями.
4. *Поддержка производителями коммерческих продуктов.* Уровень поддержки, оказываемой производителями коммерческих продуктов, варьируется в широких пределах, поскольку эти системы распространяются свободно. Поддержка производителей особенно важна в тех случаях, когда у разработчиков возникают проблемы, связанные с получением доступа к исходному коду и к подробной документации системы. Несмотря на то что производитель берет на себя обязательства по поддержке своих систем, изменение ситуации на рынке и экономических условий может привести к тому, что ему станет трудно продолжать выполнение взятых обязательств. Например, производитель коммерческой системы решил больше не поддерживать развитие какого-либо продукта из-за ограниченного спроса или, возможно, передал его другой компании, которая не хочет поддерживать все его продукты.

Конечно, маловероятно, что все эти проблемы возникнут в каждом случае использования коммерческих продуктов. По моим приблизительным подсчетам, во многих программных проектах, интегрирующих коммерческие системы, может появиться по крайней мере одна из перечисленных проблем. Соответственно преимущества в стоимости и времени выполнения работ по использованию коммерческих продуктов окажутся меньше, чем предполагалось в первоначальном оптимистическом варианте.

Все перечисленные проблемы являются проблемами жизненного цикла ПО и влияют только на начальную разработку системы. Но во многих случаях при использовании коммерческих продуктов расходы на сопровождение и модернизацию систем также могут возрасти [42], поскольку люди, участвующие в обслуживании системы, со временем все больше отдаляются от разработчиков исходной системы.

Несмотря на все эти проблемы, преимущества, получаемые при использовании коммерческих продуктов, весьма существенны, так как в этом случае можно сэкономить месяцы, а иногда и годы на разработке системы. Так как быстрое создание систем является одним из ключевых факторов для большинства программных проектов, данный вид повторного использования компонентов, вероятно, получит со временем широкое практическое применение.

14.1.3. Разработка повторно используемых компонентов

Разработка идеального компонента для повторного использования должна быть процессом (основанным на опыте и знаниях о проблемах повторного использования) создания обобщенных компонентов, которые можно адаптировать для разных вариантов их использования.

Программный компонент, предназначенный для повторного использования, имеет ряд особенностей.

1. Должен отражать стабильные абстракции предметной области, т.е. фундаментальные понятия области приложения, которые меняются медленно. Например, в банковской системе абстракциями предметной области могут быть счета, форма вкладчика, бюллетени и т.п.
2. Должен скрывать способ представления своего состояния и предоставлять операции, которые позволяют обновлять состояния и получать к нему доступ. Например, в компоненте, который представляет счет в банке, должны быть операции, позволяющие выполнить запросы по остаткам на счетах, по изменениям в остатках счета, записать операции (транзакции) на счетах и т.п.
3. Должен быть максимально независимым. В идеале компонент должен быть настолько автономным, чтобы не нуждаться в других компонентах. В действительности такое выполнимо только для совсем простых компонентов, более сложные всегда зависят от других компонентов. Лучше всего имеющиеся зависимости свести к минимуму, особенно если они связаны с такими компонентами, как изменяемые функции операционной системы.
4. Все исключительные ситуации должны быть частью интерфейса компонента. Компоненты не должны сами обрабатывать исключения, так как в разных приложениях существуют разные требования для обработки исключительных ситуаций. Лучше определить те исключения, которые необходимо обрабатывать, и объявить их как часть интерфейса компонента. Например, простой компонент, реализующий структуру данных стека, должен определять и объявить исключениями переполнение и обнуление стека.

В большинстве существующих систем имеются большие сегменты кода, которые реализуют абстракции предметной области, однако их нельзя непосредственно использовать как компоненты. Причина в несоответствии программного кода модели, показанной на рис. 14.2, четко определенному интерфейсу запросов и поставщиков сервисов. Чтобы повторно использовать такие компоненты, как правило, необходимо построить упаковщик (программное средство для создания оболочки и стандартизации внешних обращений). Упаковщик скрывает исходный код и предоставляет интерфейс для внешних компонентов, открывающий доступ к предоставляемым сервисам.

При создании компонентов, предназначенных для повторного использования, предполагается предоставление очень общего интерфейса с операциями, которые обеспечивают разные способы использования компонентов. Чтобы сделать компоненты практичными в использовании, требуется минимальный интерфейс, простой для понимания. С другой стороны, предполагаемая возможность повторного использования усложняет компоненты и потому уменьшает их понятность. Поэтому разработчики компонентов, предназначенных для повторного использования, должны прийти к некоторому компромиссу между обобщенностью и понятностью компонентов.

14.2. Семейства приложений

Один из наиболее эффективных подходов к повторному использованию базируется на понятии семейства приложений. Семейство приложений, или серия программных продуктов, – это набор приложений, имеющих общую архитектуру, отражающую специфику конкретной предметной области (см. главу 10). Вместе с тем все приложения одной серии различны. Каждый раз при создании нового приложения повторно используется общее ядро семейства приложений. Далее в процессе разработки создается несколько дополнительных компонентов, а некоторые компоненты адаптируются согласно новым требованиям.

Существуют различные специализации семейств приложений, приведем некоторые из них.

1. *Платформенная специализация*, при которой для разных платформ разрабатываются свои версии приложения. Например, приложение может иметь версии для платформ Windows NT, Solaris или Linux. В данном случае функциональность приложения обычно не меняется; подвергаются изменениям только те компоненты, которые отвечают за взаимодействие с аппаратными средствами и операционной системой.
2. *Конфигурационная специализация*, при которой разные версии приложения создаются для управления различными периферийными устройствами. Например, разные версии системы безопасности могут зависеть от типа используемой радиосистемы. В этом случае изменяется функциональность приложения для того, чтобы соответствовать периферийным устройствам, и необходимо изменить те компоненты, которые связаны с периферийными устройствами.
3. *Функциональная специализация*, при которой создаются разные версии приложения для заказчиков с различными требованиями. Например, система автоматизации библиотек может иметь несколько модификаций в зависимости от того, где она применяется – в публичной, справочной или университетской библиотеке. В этом случае изменяются компоненты, реализующие функциональность системы, и добавляются новые компоненты.

Чтобы наглядно представить эту технологию повторного использования, рассмотрим архитектуру системы управления ресурсами, изображенную на рис. 14.7. Подобные системы используются в организациях для отслеживания активов (ресурсов, запасов) и управ-

ления ими. Например, система управления ресурсами энергосистемы должна отслеживать все стационарные энергообъекты и соответствующее оборудование. В университетах система управления ресурсами может отслеживать оборудование, используемое в учебных лабораториях.

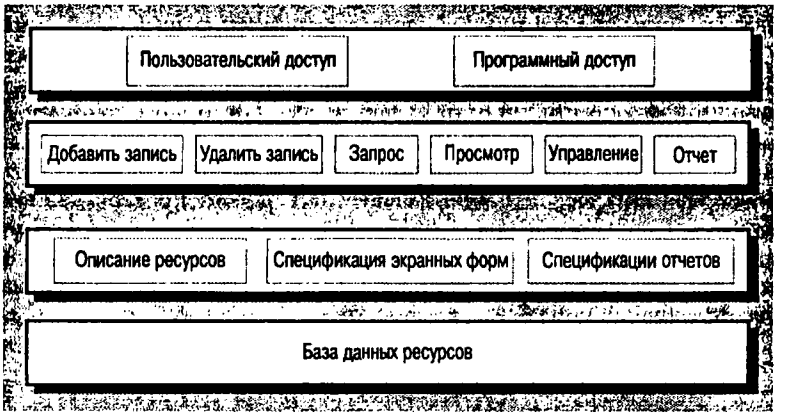


Рис. 14.7. Обобщенная система управления ресурсами

Очевидно, системы управления ресурсами будут отличаться друг от друга в зависимости от типа ресурсов и от информации, необходимой для управления ими. Например, в приложении для энергосистемы не нужны средства, позволяющие изменять размещение ресурсов, так как все объекты энергосистемы стационарны. Однако система учета ресурсов для университета должна иметь такую возможность, так как оборудование может переходить из одной лаборатории в другую.

Тем не менее все эти системы должны предоставлять основные средства для управления ресурсами: возможность добавления и удаления ресурсов, формирование запросов, просмотр базы данных ресурсов и формирование отчетов. Следовательно, архитектура систем управления ресурсами будет одинакова для целого семейства приложений, в котором отдельные приложения поддерживают разные типы ресурсов.

Для того чтобы повторное использование систем было эффективным, на этапе создания архитектуры необходимо отделить основные средства системы от конкретной информации об управляемых ресурсах и от доступа пользователей к этой информации. На рис. 14.7 разделение достигнуто благодаря многоуровневой архитектуре, в которой на одном уровне встроены описания ресурсов, формирование экранных форм и отчетов. Верхние уровни системы используют эти описания в своих методах и не содержат конкретной информации о ресурсах. Посредством изменения уровня описаний можно создавать различные приложения управления ресурсами.

Конечно, такой тип систем можно выполнить в виде объектно-ориентированных, определив сначала объект абстрактного ресурса, а затем с помощью наследования – объект, зависящий от типа управляемого ресурса. В итоге такая архитектура будет немногим отличаться от архитектуры, изображенной на рис. 14.7. Однако для систем данного типа объектно-ориентированный подход не годится. Когда приложения рассчитаны на большие базы данных, содержащие миллионы записей, но относительно малое количество типов логических модулей (соответствующих объектам и сущностям), очевидно, что объектно-ориентированная система работает менее эффективно, чем системы с реляционными базами данных. На момент написания книги коммерческие объектно-ориенти-

рованные базы все еще остаются относительно медленными и не способными поддерживать сотни транзакций в секунду.

Подобно тому как посредством описаний новых ресурсов можно создавать новые члены семейства приложений, с помощью включения новых модулей на системном уровне в систему можно добавить новые функциональные возможности. Для создания библиотечной системы (рис. 14.8) я адаптировал систему управления ресурсами, показанную на рис. 14.7. В результате в систему добавлены новые возможности для выдачи и возврата ресурсов и регистрации пользователей системой. На рис. 14.8 эти средства расположены справа. Так как программный доступ здесь не нужен, самый верхний уровень системы поддерживает доступ к ресурсам только на уровне пользователя.

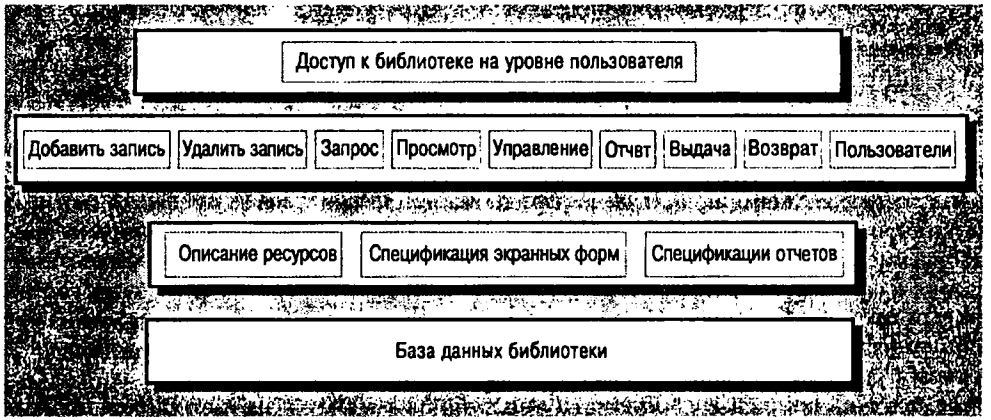


Рис. 14.8. Библиотечная система

В целом адаптация версии приложения в процессе его разработки приводит к тому, что значительная часть кода приложения используется повторно. Более того, накопленный опыт часто можно использовать для разработки других систем, поэтому объединение разработчиков ПО в отдельную группу сокращает процесс их обучения. Так как тесты для большинства компонентов приложения также можно использовать повторно, то полное время, необходимое на разработку приложения, значительно уменьшается.

Процесс адаптации семейства приложений для создания нового приложения состоит из нескольких этапов, представленных на рис. 14.9. Детали процесса могут значительно отличаться для разных прикладных областей и для различных организаций. Обобщенный процесс создания нового приложения состоит из следующих этапов.

1. *Определение требований для нового приложения.* Данный этап – обычный процесс разработки требований. Но так как система уже существует, естественно провести экспериментирование с ней и выявить те системные требования, которые необходимо изменить.
2. *Выбор наиболее подходящего члена семейства приложений.* Выполняется анализ требований, после чего выбирается наиболее подходящий член семейства, требующий внесения минимальных изменений.
3. *Пересмотр требований.* Как правило, появляется дополнительная информация, требующая внесения изменений в существующую систему, поэтому пересмотр требований на этом этапе позволяет уменьшить количество необходимых изменений.

4. *Адаптация выбранной системы.* Для системы разрабатываются новые модули, а существующие адаптируются к новым требованиям.
5. *Создание нового члена семейства приложений.* Для заказчика создается новый член семейства приложений. На этом этапе выполняется документирование ключевых особенностей системы, чтобы в дальнейшем ее можно было использовать как основу для разработки других систем.

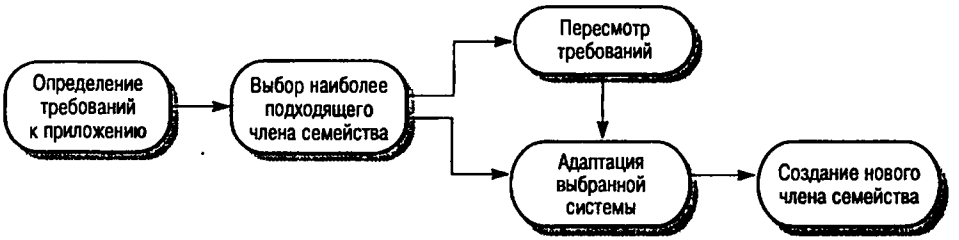


Рис. 14.9. Процесс разработки нового члена семейства приложений

В процессе создания нового члена семейства приложений часто требуется найти некий компромисс между наиболее полным использованием существующих приложений и выполнением конкретных требований для нового приложения. Чем более детальны требования к системе, тем меньше вероятность, что имеющиеся компоненты будут соответствовать этим требованиям. Но практически всегда можно достигнуть определенного компромисса и ограничить объем изменений, вносимых в систему, тогда процесс создания новой системы выполняется быстро и с небольшими затратами.

За редким исключением, семейства приложений создаются из имеющихся приложений, т.е. организация создает приложения и затем при необходимости использует их как основу для разработки нового приложения. Но вместе с тем внесение изменений нарушает структуру приложения, поэтому рано или поздно принимается решение о создании семейства обобщенных приложений. В этом случае активно используются знания, собранные в процессе создания исходной группы приложений.

14.3. Проектные паттерны

Попытки повторно использовать действующие компоненты постоянно ограничиваются конкретными решениями, принятыми системными разработчиками. Решения могут относиться к отдельным алгоритмам, используемым при реализации компонентов, к объектам и к типам данных в интерфейсах компонентов. Если решения противоречат конкретным требованиям к компонентам, то повторное использование компонентов либо невозможно, либо делает систему неэффективной.

Один из способов решения данной проблемы — повторное использование более абстрактных структур, не содержащих деталей реализации. Такие структуры разрабатываются специально для того, чтобы соответствовать определенному приложению. Первые реализации этого подхода привели к документированию и опубликованию фундаментальных алгоритмов [201], а затем к документированию абстрактных типов данных, таких как стеки, деревья и списки [53]. Совсем недавно такой способ повторного использования обобщен в понятии паттерна.

Проектные паттерны (design patterns) [124] появились из идей, выдвинутых Кристофером Александером (Alexander, [8]), который предложил удобные и эффективные обобщенные паттерны разработки конкретных проектов. Паттерн — это описание про-

блемы и метода ее решения, позволяющее в дальнейшем использовать это решение в разных условиях. Паттерн не является детальной спецификацией. Скорее, он представляет собой описание, в котором аккумулированы знания и опыт. Паттерн – гарантированное решение общей проблемы.

При создании ПО проектные паттерны всегда связаны с объектно-ориентированным проектированием. Чтобы обеспечить всеобщность, паттерны часто используют такие объектно-ориентированные понятия, как наследование и полиморфизм. Однако общий принцип использования паттернов одинаково применим при любом подходе к проектированию ПО.

В [124] определены четыре основных элемента проектного паттерна.

1. Содержательное имя, которое является ссылкой на паттерн.
2. Описание проблемной области с перечислением всех ситуаций, в которых можно использовать паттерн.
3. Описание решений с отдельным описанием различных частей решения и их взаимоотношений. Это не описание конкретного проекта, а *шаблон* проектных решений, который можно использовать различными способами. В описании решений часто используются графические представления, которые показывают взаимоотношения между объектами и классами объектов в данном решении.
4. Описание "выходных" результатов – это описание результатов и компромиссов, необходимых для применения паттерна. Обычно используется для того, чтобы помочь разработчикам оценить конкретную ситуацию и выбрать для нее наиболее подходящий паттерн.

Часто в описание паттерна вводятся также разделы *мотивации* (обоснование полезности паттерна) и *применимости* (описание ситуаций, в которых можно использовать паттерн).

В качестве примера рассмотрим один из наиболее часто используемых паттернов, предложенных в работе [124], а именно Обозреватель (Observer) (см. врезку 14.1). Данный паттерн используется тогда, когда необходимы разные представления состояния объекта. Он выделяет нужный объект и представляет его в разных формах; на рис. 14.10 показаны два графических представления одного и того же набора данных.

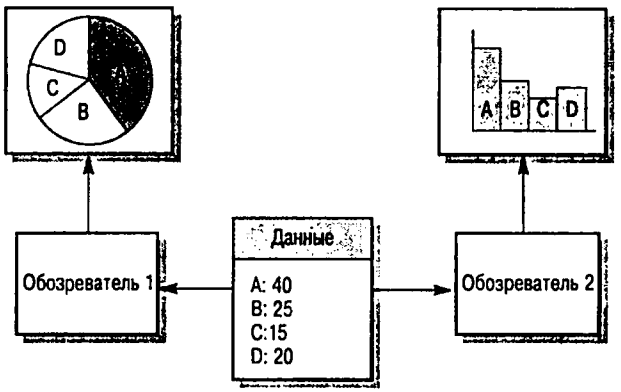


Рис. 14.10. Различные представления данных

Врезка 14.1. Описание паттерна Обозреватель

Имя паттерна. Обозреватель

Описание. Отделяет отображение состояния объекта от самого объекта и предлагает различные способы представления состояния. При изменении состояния объекта все представления автоматически обновляются, чтобы отобразить происшедшие изменения.

Описание проблемы. Во многих ситуациях требуется представить информацию о состоянии некоторого объекта несколькими разными способами, например используя графическое и табличное представления. Все представления взаимосвязаны и должны обновляться при изменении состояния.

Данный паттерн можно использовать во всех ситуациях, где требуется несколько разных представлений информации о состоянии объекта и где нет необходимости знать форматы представления данных о состоянии объекта.

Описание решения. Структура паттерна показана на рис. 14.11. В нем определены два абстрактных объекта: Subject (Данные) и Observer (Обозреватель), а также два конкретных объекта: ConcreteSubject (Конкретные данные) и ConcreteObserver (Конкретный обозреватель), которые наследуют свойства соответствующих абстрактных объектов. Отображаемое состояние поддерживается объектом ConcreteSubject, который также наследует методы от Subject, позволяющие ему добавлять и удалять объекты Observer (методы Attach и Detach) и выдавать оповещение при изменении состояния (метод Notify).

Объект ConcreteObserver обрабатывает копию состояния ConcreteSubject (копию subjectState, полученную с помощью метода GetState (Получить состояние)) и реализует метод Update (Обновить) интерфейса Observer, который позволяет сохранять копии состояния. ConcreteObserver автоматически отображает это состояние.

Результаты. Для оптимизации обозревателя необходима дополнительная информация об объектах. Изменения в формате отображаемых данных вызовут серию связанных изменений в созданных обозревателях.

Обычно в паттернах классы объектов и взаимоотношения между ними изображаются с помощью специальных графических нотаций. На рис. 14.11 представлен паттерн Обозреватель в нотации языка UML.

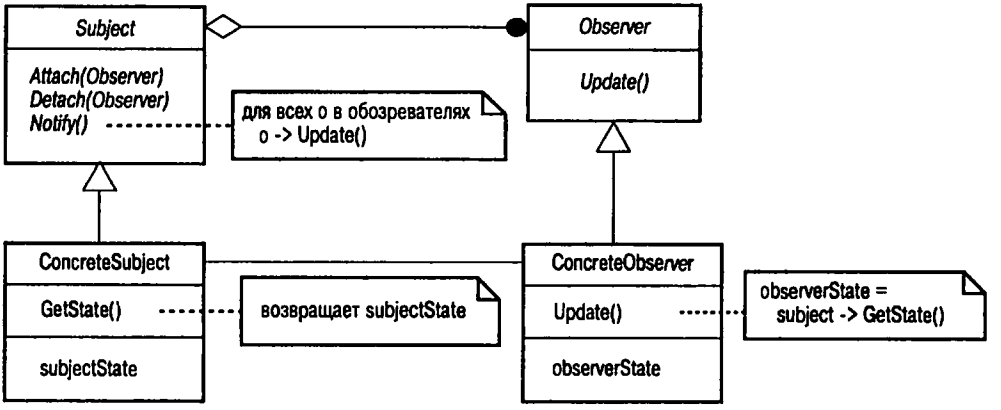


Рис. 14.11. Паттерн Обозреватель

Применение паттернов является весьма эффективным способом повторного использования; однако, по моему мнению, данный метод требует значительных затрат на освоение и может эффективно использоваться в проектировании систем только опытными программистами. Причина кроется в достаточно высокой сложности паттернов. Паттерн не похож на исполняемый компонент, для использования которого достаточно изучить только его интерфейс. Очевидно, что на изучение паттерна требуется определенное время.

Использовать паттерны могут только опытные программисты, поскольку лишь они способны распознать общие ситуации, в которых можно применить тот или иной паттерн. Неопытные программисты, даже если они прочитали несколько книг, описывающих паттерны, на практике зачастую не могут определить, где следует использовать паттерн, а где необходимо нестандартное специальное решение.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Проектирование с повторным использованием компонентов означает проектирование программных систем с учетом уже имеющихся компонентов ПО.
- К преимуществам повторного использования программного обеспечения можно отнести более низкие затраты, более быструю разработку и пониженные риски. Также повышается надежность систем и появляется возможность более эффективно применять опыт и знания специалистов, привлекая их к проектированию повторно используемых компонентов.
- Покомпонентная разработка ПО основывается на использовании компонентов с четко определенными интерфейсами запросов и поставщиков сервисов без конкретизации знаний о внутреннем устройстве компонентов. Можно повторно использовать разные типы компонентов: функции, абстракции данных, структуры и законченные системы приложений.
- Под повторным использованием коммерческих продуктов понимается повторное использование крупномодульных готовых программных систем. Применение коммерческих продуктов может значительно уменьшить расходы и время на разработку нового ПО.
- Программные компоненты, создаваемые для повторного использования, должны быть независимыми, отображать абстракции предметной области, предоставлять доступ к состоянию через методы интерфейса и не должны сами обрабатывать исключительные ситуации.
- Семейство приложений — это группа приложений, которые разрабатываются на основе одного или нескольких базовых приложений. Базовая обобщенная система адаптируется для соответствия требованиям разрабатываемой системы.
- Проектные паттерны — это абстракции высокого уровня, которые документируют успешные проектные решения. Они являются основой при повторном использовании проектных решений в объектно-ориентированных разработках. Описание паттерна содержит имя паттерна, описание проблемы и решения, а также результаты и компромиссы использования шаблона.

Упражнения

14.1. Каковы основные технические и нетехнические факторы, затрудняющие повторное использование программного обеспечения?

14.2. Объясните, почему сокращение расходов при повторном использовании компонентов не прямо пропорционально размерам повторно используемых компонентов.

14.3. Приведите четыре аргумента против повторного использования компонентов.

14.4. Предположите возможные интерфейсы запросов и поставщиков сервисов для следующих компонентов.

- Компонент, реализующий счет в банке.
- Компонент, реализующий не зависящую от языка клавиатуру. Клавиатуры в разных странах имеют различную организацию клавиш и разные наборы символов.
- Компонент, реализующий средство управления версиями, рассмотренное в главе 29.

14.5. Чем отличается повторное использование объектной структуры приложения от повторного использования коммерческих продуктов? Почему иногда проще повторно использовать коммерческий продукт, чем объектную структуру приложения?

14.6. На примере метеорологической станции, описанной в главе 12, предложите архитектуру семейства приложений, которые связаны с удаленными наблюдением и сбором метеоданных.

14.7. На примере семейства приложений по управлению ресурсами (см. рис. 14.7) подумайте, какие методы необходимо добавить или изменить, чтобы можно было делать повторный заказ на отдельные виды ресурсов, если их количество становится меньше некоторой заданной величины.

14.8. Почему паттерны — эффективный способ повторного использования в проектировании? Каковы недостатки этого подхода?

14.9. Повторное использование увеличивает количество вопросов о собственности, охраняемой авторским и интеллектуальным правом. Если заказчик оплачивает разработчику ПО заказ на разработку какой-либо системы, кто имеет право повторно использовать созданный код? Имеет ли право разработчик использовать этот код в качестве основы для базового компонента? Какими должны быть механизмы оплаты труда разработчика повторно используемых компонентов? Обсудите эти и другие этические вопросы, связанные с повторным использованием программного обеспечения.

15

Проектирование интерфейса пользователя

Цели

Цель настоящей главы — познакомить с основными аспектами проектирования интерфейса пользователя, которые должны знать разработчики ПО. Прочитав эту главу, вы должны:

- знать основные принципы проектирования интерфейса пользователя;
- освоить пять разных стилей взаимодействия пользователя с программными системами;
- знать разные стили представления информации и то, в каких случаях целесообразно графическое представление данных;
- познакомиться с основными правилами проектирования средств поддержки пользователя, встроенных в программное обеспечение;
- иметь представление об основных показателях удобства использования систем.

Содержание

Проектирование вычислительных систем охватывает широкий спектр проектных действий – от проектирования аппаратных средств до проектирования интерфейса пользователя. Организации-разработчики часто нанимают специалистов для проектирования аппаратных средств и очень редко для проектирования интерфейсов. Таким образом, специалистам по разработке ПО зачастую приходится проектировать и интерфейс пользователя. Если в больших компаниях в этот процесс вовлекаются специалисты по инженерной психологии, то в небольших компаниях услугами таких специалистов практически не пользуются.

Грамотно спроектированный интерфейс пользователя крайне важен для успешной работы системы. Сложный в применении интерфейс, как минимум, приводит к ошибкам пользователя. Иногда они просто отказываются работать с программной системой, несмотря на ее функциональные возможности. Если информация представляется сбивчиво или непоследовательно, пользователи могут понять ее неправильно, в результате чего их последующие действия могут привести к повреждению данных или даже к сбою в работе системы.

В 1982 году, во время выхода первой редакции этой книги, стандартным устройством взаимодействия между пользователем и программой был "беззвучный" буквенно-цифровой (текстовый) терминал, отображающий на черном поле символы зеленого или синего цвета. В то время интерфейсы пользователя были текстовыми или создавались в виде специальных форм. Сейчас почти все пользователи работают на персональных компьютерах. Все современные персональные компьютеры поддерживают графический интерфейс пользователя (graphical user interface – GUI), который подразумевает использование цветного графического экрана с высоким разрешением и позволяет работать с мышью и с клавиатурой.

Хотя текстовые интерфейсы еще достаточно широко применяются, особенно в наследуемых системах, в наше время пользователи предпочитают работать с графическим интерфейсом. В табл. 15.1 перечислены основные элементы GUI.

Таблица 15.1. Элементы графических интерфейсов пользователя

Элементы	Описание
Окна	Позволяют отображать на экране информацию разного рода
Пиктограммы	Представляют различные типы данных. В одних системах пиктограммы представляют файлы, в других – процессы
Меню	Ввод команд заменяется выбором команд из меню
Указатели	Мышь используется как устройство указания для выбора команд из меню и для выделения отдельных элементов в окне
Графические элементы	Могут использоваться совместно с текстовыми

Графические интерфейсы обладают рядом преимуществ.

1. Их относительно просто изучить и использовать. Пользователи, не имеющие опыта работы с компьютером, могут легко и быстро научиться работать с графическим интерфейсом.
2. Каждая программа выполняется в своем окне (экране). Можно переключаться из одной программы в другую, не теряя при этом данные, полученные в ходе выполнения программ.
3. Режим полноэкранного отображения окон дает возможность прямого доступа к любому месту экрана.

Цель данной главы — привлечь внимание разработчиков ПО к некоторым ключевым проблемам, лежащим в основе проектирования интерфейсов пользователя. Разработчики и программисты обычно компетентны в использовании таких технологий, как классы Swing в языке Java [103] или HTML [249], являющиеся основой реализации интерфейсов пользователя. Однако эту технологию далеко не всегда применяют надлежащим образом, в результате чего интерфейсы пользователя получаются неэлегантными, неудобными и сложными в использовании.

В этой главе я приведу несколько рекомендаций по проектированию средств конечного пользователя, не рассматривая весь процесс проектирования этих средств. Из-за нехватки места рассматриваются только графические интерфейсы. Специальные интерфейсы, например для мобильных телефонов, телевизионных приемников, копировальной техники или факсимильных аппаратов, рассматриваться не будут. Здесь я сделаю только краткое введение в тему проектирования интерфейсов пользователя. Дополнительную информацию по данной теме можно найти в книгах [316, 99, 281].

На рис. 15.1 изображен итерационный процесс проектирования пользовательского интерфейса. Как отмечалось в главе 8, наиболее эффективным подходом к проектированию интерфейса пользователя является разработка с применением моделирования пользовательских функций. В начале процесса прототипирования создаются бумажные макеты интерфейса, затем разрабатываются экранные формы, моделирующие взаимодействие с пользователем. Желательно, чтобы конечные пользователи принимали активное участие в процессе проектирования интерфейса [258]. В одних случаях пользователи помогут оценить интерфейс; в других будут полноправными членами проектной группы [207, 138].

Рис. 15.1. Процесс проектирования интерфейса пользователя

Важным этапом процесса проектирования интерфейса пользователя является анализ деятельности пользователей, которую должна обеспечить вычислительная система. Не изучив того, что, с точки зрения пользователя, должна делать система, невозможно сформировать реалистический взгляд на проектирование эффективного интерфейса. Для анализа нужно (как правило, одновременно) применять различные методики, а именно: анализ задач [94], этнографический подход (см. главу 6) [328, 167], опросы пользователей и наблюдения за их работой.

15.1. Принципы проектирования интерфейсов пользователя

Разработчики интерфейсов всегда должны учитывать физические и умственные способности людей, которые будут работать с программным обеспечением. Люди на короткое время могут запомнить весьма ограниченный объем информации (см. главу 22) и совершают ошибки, если приходится вводить вручную большие объемы данных или работать в напряженных условиях. Физические возможности людей могут существенно различаться, поэтому при проектировании интерфейсов пользователя необходимо постоянно помнить об этом.

Основой принципов проектирования интерфейсов пользователя являются человеческие возможности. В табл. 15.2 представлены основные принципы, применимые при проектировании любых интерфейсов пользователя. Более детальный перечень "руководящих" правил проектирования интерфейсов можно найти в книге [316].

Таблица 15.2. Принципы проектирования интерфейсов пользователя

Принцип	Описание
Учет знаний пользователя	В интерфейсе необходимо использовать термины и понятия, взятые из опыта будущих пользователей системы
Согласованность	Интерфейс должен быть согласованным в том смысле, что однотипные (но различные) операции должны выполняться одним и тем же способом
Минимум неожиданностей	Поведение системы должно быть прогнозируемым
Способность к восстановлению	Интерфейс должен иметь средства, позволяющие пользователям восстановить данные после ошибочных действий
Руководство пользователя	Интерфейс должен предоставлять необходимую информацию в случае ошибок пользователя и поддерживать средства контекстно-зависимой справки
Учет разнородности пользователей	В интерфейсе должны быть средства для удобного взаимодействия с пользователями, имеющими разный уровень квалификации и различные возможности

Принцип учета знаний пользователя предполагает следующее: интерфейс должен быть настолько удобен при реализации, чтобы пользователям не понадобилось особых усилий, чтобы привыкнуть к нему. В интерфейсе должны использоваться термины, понятные пользователю, а объекты, управляемые системой, должны быть напрямую связаны с рабочей средой пользователя. Например, если разрабатывается система, предназначенная для авиадиспетчеров, то управляемыми объектами в ней должны быть самолеты, траектории полетов, сигнальные знаки и т.п. Основную реализацию интерфейса в терминах файловых структур и структур данных необходимо скрыть от конечного пользователя.

Принцип согласованности интерфейса пользователя предполагает, что команды и меню системы должны быть одного формата, параметры должны передаваться во все команды одинаково и пунктуация команд должна быть схожей. Такие интерфейсы сокращают время на обучение пользователей. Знания, полученные при изучении какой-либо команды или части приложения, можно затем применить при работе с другими частями системы.

В данном случае речь идет о согласованности низкого уровня. И создатели интерфейса всегда должны стремиться к нему. Однако желательна согласованность и более высокого уровня. Например, удобно, когда для всех типов объектов системы поддерживаются одинаковые методы (такие, как печать, копирование и т.п.). Однако полная согласованность невозможна и даже нежелательна [140]. Например, операцию удаления объектов рабочего стола целесообразно реализовать посредством их перетаскивания в корзину. Но в текстовом редакторе такой способ удаления фрагментов текста кажется неестественным.

Всегда нужно соблюдать следующий принцип: количество неожиданностей должно быть минимальным, так как пользователей раздражает, когда система вдруг начинает вести себя непредсказуемо. При работе с системой у пользователей формируется определенная модель ее функционирования. Если его действие в одной ситуации вызывает определенную реакцию системы, естественно ожидать, что такое же действие в другой ситуации приведет к аналогичной реакции. Если же происходит совсем не то, что ожидалось, пользователь либо удивляется, либо не знает, что делать. Поэтому разработчики интерфейсов должны гарантировать, что схожие действия произведут похожий эффект.

Очень важен принцип восстанавливаемости системы, так как пользователи всегда допускают ошибки. Правильно спроектированный интерфейс может уменьшить количество ошибок пользователя (например, использование меню позволяет избежать ошибок, которые возникают при вводе команд с клавиатуры), однако все ошибки устранить невозможно. В интерфейсах должны быть средства, по возможности предотвращающие ошибки пользователя, а также позволяющие корректно восстановить информацию после ошибок. Эти средства бывают двух видов.

1. *Подтверждение деструктивных действий.* Если пользователь выбрал потенциально деструктивную операцию, то он должен еще раз подтвердить свое намерение.
2. *Возможность отмены действий.* Отмена действия возвращает систему в то состояние, в котором она находилась до их выполнения. Не лишней будет поддержка многоуровневой отмены действий, так как пользователи не всегда сразу понимают, что совершили ошибку.

Следующий принцип – поддержка пользователя. Средства поддержки пользователей должны быть встроены в интерфейс и систему и обеспечивать разные уровни помощи и справочной информации. Должно быть несколько уровней справочной информации – от основ для начинающих до полного описания возможностей системы. Справочная система должна быть структурированной и не перегружать пользователя излишней информацией при простых запросах к ней (см. раздел 15.4).

Принцип учета разнородности пользователей предполагает, что с системой могут работать разные их типы. Часть пользователей работает с системой нерегулярно, время от времени. Но существует и другой тип – "опытные пользователи", которые работают с приложением каждый день по несколько часов. Случайные пользователи нуждаются в таком интерфейсе, который "руководил" бы их работой с системой, в то время как опытным пользователям требуется интерфейс, который позволил бы им максимально быстро взаимодействовать с системой. Кроме того, поскольку некоторые пользователи могут иметь разные физические недостатки, в интерфейсе должны быть средства, которые помогли бы им перенастроить интерфейс под себя. Это могут быть средства, позволяющие отображать увеличенный текст, замещать звук текстом, создавать кнопки больших размеров и т.п.

Принцип признания многообразия категорий пользователей может противоречить другим принципам проектирования интерфейсов, например согласованности интерфейса. Аналогично, необходимый уровень справочной информации для разных типов поль-

зователей может радикально отличаться. Невозможно создать такую справочную систему, которая подошла бы всем пользователям. Разработчик интерфейса должен всегда быть готовым к компромиссным решениям в зависимости от реальных пользователей системы.

15.2. Взаимодействие с пользователем

Разработчику интерфейса пользователя вычислительных систем необходимо решить две главные задачи: каким образом пользователь будет вводить данные в систему и как данные будут представлены пользователю. "Правильный" интерфейс должен обеспечивать и взаимодействие с пользователем, и представление информации.

В этом разделе обсуждаются вопросы взаимодействия системы с пользователем. Представление данных рассматривается в разделе 15.3. Интерфейс пользователя обеспечивает ввод команд и данных в вычислительную систему. На первых вычислительных машинах был только один способ ввода данных — через интерфейс командной строки, причем для взаимодействия с машиной использовался специальный командный язык. Такой способ годился только для опытных пользователей, поэтому позже были разработаны более упрощенные способы ввода данных. Все эти виды взаимодействия можно отнести к одному из пяти основных стилей взаимодействия [316].

1. *Непосредственное манипулирование.* Пользователь взаимодействует с объектами на экране. Например, для удаления файла пользователь просто перетаскивает его в корзину.
2. *Выбор из меню.* Пользователь выбирает команду из списка пунктов меню. Очень часто выбранная команда воздействует только на тот объект, который выделен (выбран) на экране. При таком подходе для удаления файла пользователь сначала выбирает файл, а затем команду на удаление.
3. *Заполнение форм.* Пользователь заполняет поля экранной формы. Некоторые поля могут иметь свое меню (выпадающее меню или списки). В форме могут быть командные кнопки, при щелчке мышью на которых инициируют некоторое действие. Чтобы удалить файл с помощью интерфейса, основанного на форме, надо ввести в поле формы имя файла и затем щелкнуть на кнопке удаления, присутствующей в форме.
4. *Командный язык.* Пользователь вводит конкретную команду с параметрами, чтобы указать системе, что она должна дальше делать. Чтобы удалить файл, пользователь вводит команду удаления с именем файла в качестве параметра этой команды.
5. *Естественный язык.* Пользователь вводит команду на естественном языке. Чтобы удалить файл, пользователь может ввести команду "удалить файл с именем XXX".

Каждый из этих стилей взаимодействия имеет преимущества и недостатки и наилучшим образом подходит разным типам приложений и различным категориям пользователей [316]. В табл. 15.3 перечислены основные преимущества и недостатки перечисленных стилей взаимодействия и указаны типы приложений, в которых они обычно используются.

Конечно, стили взаимодействия редко используются в чистом виде, в одном приложении может использоваться одновременно несколько разных стилей. Например, в операционной системе Microsoft Window поддерживается несколько стилей: прямое манипулирование пиктограммами, представляющими файлы и папки, выбор команд из меню, ручной ввод некоторых команд, таких как команды конфигурирования системы, использование форм (диалоговых окон).

Таблица 15.3. Преимущества и недостатки стилей взаимодействия пользователя с системой

Стиль взаимодействия	Основные преимущества	Основные недостатки	Примеры приложений
Прямое манипулирование	Быстрое и интуитивно понятное взаимодействие. Легок в изучении	Сложная реализация. Подходит только там, где есть зрительный образ задачи и объектов	Видеоигры; системы автоматического проектирования
Выбор из меню	Сокращение количества ошибок пользователя. Ввод с клавиатуры минимальный	Медленный вариант для опытных пользователей. Может быть сложным, если меню состоит из большого количества вложенных пунктов	Главным образом системы общего назначения
Заполнение форм	Простой ввод данных. Легок в изучении	Занимает пространство на экране	Системы управления запасами; обработка финансовой информации
Командный язык	Мощный и гибкий	Труден в изучении. Сложно предотвратить ошибки ввода	Операционные системы; библиотечные системы
Естественный язык	Подходит неопытным пользователям. Легко настраивается	Требует большого ручного набора	Системы расписания; системы хранения данных WWW

Пользовательские интерфейсы приложений World Wide Web базируются на средствах, предоставляемых языком HTML (язык разметки Web-страниц) вместе с другими языками, например Java, который связывает программы с компонентами Web-страниц. В основном интерфейсы Web-страниц проектируются для случайных пользователей и представляют собой интерфейсы в виде форм. В Web-приложениях можно создавать интерфейсы, в которых применялся бы стиль прямого манипулирования, однако к моменту написания книги проектирование таких интерфейсов представляло достаточно сложную в аспекте программирования задачу.

В принципе необходимо применять различные стили взаимодействия для управления разными системными объектами. Данный принцип составляет основу модели Сихейма (Seeheim) пользовательских интерфейсов [278]. В этой модели разделяются представление информации, управление диалоговыми средствами и управление приложением. На самом деле такая модель является скорее идеальной, чем практической, однако почти всегда есть возможность разделить интерфейсы для разных классов пользователей (например, начинающих и опытных). На рис. 15.2 изображена подобная модель с разделенными интерфейсом командного языка и графическим интерфейсом, лежащая в основе некоторых операционных систем, в частности Linux.

Рис. 15.2. Множественный интерфейс

Разделение представления, взаимодействия и объектов, включенных в интерфейс пользователя, является основным принципом подхода "модель–представление–контроллер", который обсуждается в следующем разделе. Эта модель сравнима с моделью Сихейма, однако используется при реализации отдельных объектов интерфейса, а не всего приложения.

15.3. Представление информации

В любой интерактивной системе должны быть средства для представления данных пользователям. Данные в системе могут отображаться по-разному: например, вводимая информация может отображаться непосредственно на дисплее (как, скажем, текст в текстовом редакторе) или преобразовываться в графическую форму. Хорошим тоном при проектировании систем считается отделение представления данных от самих данных. До некоторой степени разработка такого ПО противоречит объектно-ориентированному подходу, при котором методы, выполняемые над данными, должны быть определены самими данными. Однако в нашем случае предполагается, что разработчик объектов всегда знает наилучший способ представления данных; хотя это, конечно, не всегда так. Часто определить наилучший способ представления данных конкретного типа довольно трудно, в таком случае объектные структуры не должны быть "жесткими".

После того как представление данных в системе отделено от самих данных, изменения в представлении данных на экране пользователя происходят без изменения самой системы (рис. 15.3).

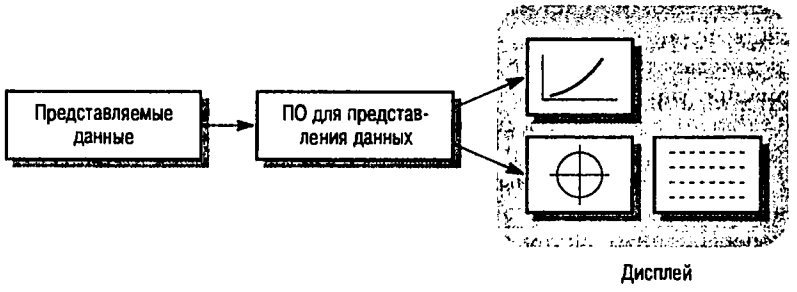


Рис. 15.3. Представление данных

Подход "модель–представление–контроллер" (МПК), представленный на рис. 15.4, получил первоначальное применение в языке Smalltalk как эффективный способ поддержки различных представлений данных [130]. Пользователь может взаимодействовать с каждым типом представления. Отображаемые данные инкапсулированы в объекты модели. Каждый объект модели может иметь несколько отдельных объектов представлений, где каждое представление – это разные отображения модели. Я уже иллюстрировал этот подход в предыдущей главе, где рассматривались объектно-ориентированные структуры приложений.

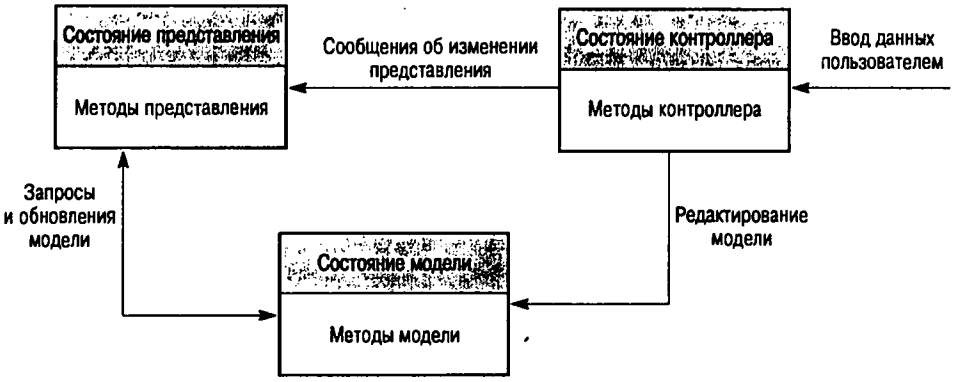


Рис. 15.4. Модель МПК взаимодействия с пользователем

Каждое представление имеет связанный с ним объект контроллера, который обрабатывает введенные пользователем данные и обеспечивает взаимодействие с устройствами. Такая модель может представить числовые данные, например, в виде диаграмм или таблиц. Модель можно редактировать, изменяя значения в таблице или параметры диаграммы. Такой подход рассмотрен в главе 14 при описании паттерна Обозреватель (раздел 14.3).

Чтобы найти наилучшее представление информации, необходимо знать, с какими данными работают пользователи и каким образом они применяются в системе. Принимая решение по представлению данных, разработчик должен учитывать ряд факторов.

1. Что нужно пользователю – точные значения данных или соотношения между значениями?
2. Насколько быстро будут происходить изменения значений данных? Нужно ли немедленно показывать пользователю изменение значений?
3. Должен ли пользователь предпринимать какие-либо действия в ответ на изменение данных?
4. Нужно ли пользователю взаимодействовать с отображаемой информацией посредством интерфейса с прямым манипулированием?
5. Информация должна отображаться в текстовом (описательно) или числовом формате? Важны ли относительные значения элементов данных?

Если данные не изменяются в течение сеанса работы с системой, их можно представить либо в графическом, либо в текстовом виде, в зависимости от типа приложения. Текстовое представление данных занимает на экране мало места, но в таком случае данные нельзя охватить одним взглядом. С помощью разных стилей представления неизменяемые данные следует от-

делить от динамически изменяющихся данных. Например, статические данные можно выделить особым шрифтом, подчеркнуть особым цветом либо обозначить пиктограммами.

Если требуется точная цифровая информация и данные изменяются относительно медленно, их можно отображать в текстовом виде. Там, где данные изменяются быстро, обычно используется графическое представление.

В качестве примера рассмотрим систему, которая помесячно записывает и подбивает итоги по данным продаж некой компании. На рис. 15.5 видно, что одни и те же данные можно представить в виде текста и в графическом виде.

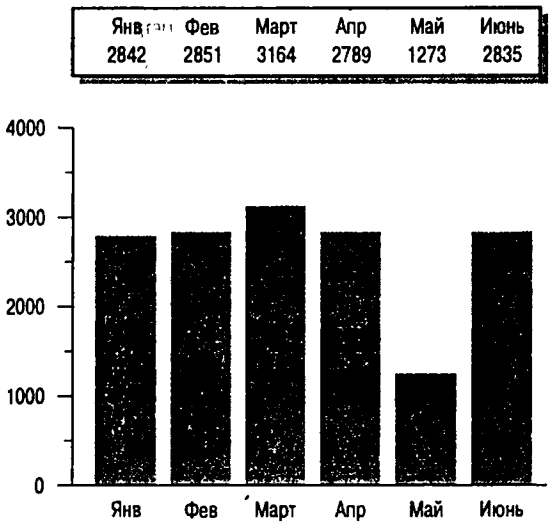


Рис. 15.5. Альтернативные представления данных

Менеджерам, изучающим данные о продажах, обычно больше нужны тенденции изменения или аномальные данные, чем их точные значения. Графическое представление этой информации в виде гистограммы позволяет выделить аномальные данные за март и май, значительно отличающиеся от остальных данных. Как видно из рис. 15.5, данные в текстовом представлении занимают меньше места, чем в графическом.

Динамические изменения числовых данных лучше отображать графически, используя аналоговые представления. Постоянно изменяющиеся цифровые экраны сбивают пользователей с толку, поскольку точные значения данных быстро не воспринимаются. Графическое отображение данных при необходимости можно дополнить точными значениями. Различные способы представления изменяющихся числовых данных показаны на рис. 15.6.

Непрерывные аналоговые отображения помогают наблюдателю оценить относительные значения данных. На рис. 15.7 числовые значения температуры и давления приблизительно одинаковы. Но при графическом отображении видно, что значение температуры близко к максимальному, в то время как значение давления не достигло даже 25% от максимума. Обычно, кроме текущего значения, наблюдателю требуется знать максимальные (или минимальные) возможные значения. Он должен в уме вычислять относительное состояние считываемых данных. Дополнительное время, необходимое для расчетов, может привести к ошибкам оператора в стрессовых ситуациях, когда возникают проблемы и на дисплее отображаются аномальные данные.

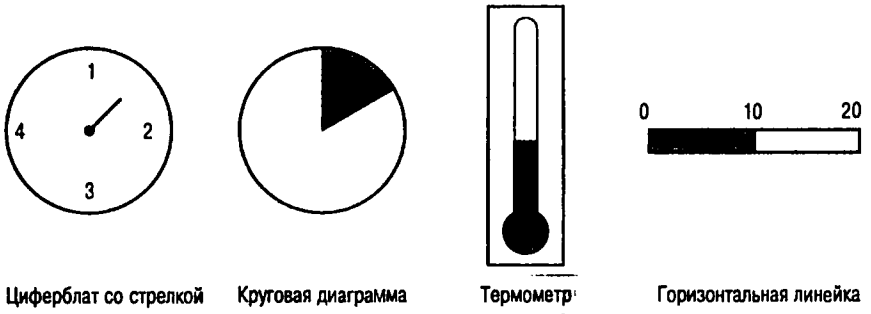


Рис.15.6. Способы представления динамически изменяющихся числовых данных

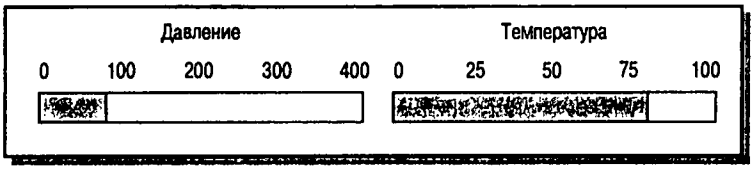


Рис. 15.7. Графическое представление данных, показывающее значения по отношению к максимальным

При представлении точных буквенно-цифровых данных для выделения особой информации можно использовать графические элементы. Вместо обычной строки данные лучше поместить в прямоугольник или отметить пиктограммой (рис. 15.8). Прямоугольник с сообщением помещается поверх текущего экрана, тем самым привлекая к нему внимание пользователя.

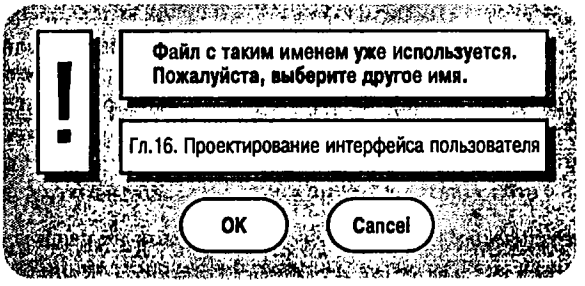


Рис. 15.8. Выделение буквенно-цифровых данных

Выделение информации с помощью графических элементов можно также использовать для привлечения внимания к изменениям, происходящим в разных частях экрана. Но, если изменения происходят очень быстро, не следует использовать графические элементы, поскольку быстрые изменения могут привести к наложению экранов, что сбивает с толку и раздражает пользователей.

При представлении больших объемов данных можно использовать разные приемы визуализации, которые указывают на родственные элементы данных. Разработчики интерфейсов должны помнить о возможностях визуализации, особенно если интерфейс системы должен отображать физические сущности (объекты). Вот несколько примеров визуализации данных.

1. Отображение метеорологических данных, собранных из разных источников, в виде метеорологических карт с изобарами, воздушными фронтами и т.п.
2. Графическое отображение состояния телефонной сети в виде связанного множества узлов.
3. Визуализация состояния химического процесса с показом давлений и температур в группе связанных между собой резервуаров и труб.
4. Модель молекулы и манипулирование ею в трехмерном пространстве посредством системы виртуальной реальности.
5. Отображение множества Web-страниц в виде дерева гипертекстовых ссылок [208].

В книге [316] содержится хорошее описание различных подходов к визуализации. Там же представлены различные классы визуализации, часто используемые на практике. В частности, к ним относится визуализация данных с использованием двух- и трехмерных деревьев и сетей. Большинство из них связаны с отображением больших объемов данных, управляемых компьютером. Наиболее широко визуализация применяется в интерфейсах для представления некоторых физических структур, например молекулярной структуры лекарства, телекоммуникационных сетей и т.п. Трехмерные представления, для создания которых используется специальное оборудование виртуальной реальности, являются особенно эффективным способом визуализации.

15.3.1. Использование в интерфейсах цвета

Во всех интерактивных системах, независимо от их назначения, поддерживаются цветные экраны, поэтому в пользовательских интерфейсах часто используются различные цвета. В некоторых системах цвета применяют в основном для выделения определенных элементов (например, в текстовых редакторах для выделения фрагментов текста); в других системах (таких, как системы автоматического проектирования) цветами обозначают разные уровни проектов.

Правильное использование цветов делает интерфейс пользователя более удобным для понимания и управления. Вместе с тем использование цветов может быть неправильным, в результате чего создаются интерфейсы, которые визуально неприглядны и даже провоцируют ошибки. Основным принципом разработчиков интерфейсов должно быть осторожное использование цветов на экранах. В работе [316] дается 14 правил эффективного использования цвета в пользовательских интерфейсах. Вот наиболее важные из них.

1. *Используйте ограниченное количество цветов.* Для окон не следует использовать более четырех или пяти разных цветов, в интерфейсе системы не должно быть более семи цветов.
2. *Используйте разные цвета для показа изменений в состоянии системы.* Если на экране изменились цвета, значит, произошло какое-то событие. Выделение цветом особенно важно в сложных экранах, в которых отображаются сотни разных объектов.
3. *Для помощи пользователю используйте цветовое кодирование.* Если пользователям необходимо выделять аномальные элементы, выделите их цветом; если требуется найти подобные элементы, выделите их одинаковым цветом.
4. *Используйте цветовое кодирование продуманно и последовательно.* Если в какой-либо части системы сообщения об ошибке отображаются, например, красным цветом, то во всех других частях подобные сообщения должны отображаться таким же цветом. Тогда красный цвет не следует использовать где-либо еще. Если же красный цвет использу-

ется еще где-то в системе, пользователь может интерпретировать появление красного цвета как сообщение об ошибке. Следует помнить, что у определенных типов пользователей имеются свои представления о значении отдельных цветов.

5. *Осторожно используйте дополняющие цвета.* Физиологические особенности человеческого глаза не позволяют одновременно сфокусироваться на красном и синем цветах. Поэтому последовательность красных и синих изображений вызывает зрительное напряжение. Некоторые комбинации цветов также могут визуально нарушать или затруднять чтение.

Чаще всего разработчики интерфейсов допускают две ошибки: привязка значения к определенному цвету и использование большого количества цветов на экране. Использовать цвета для представления значения не следует по двум причинам. Около 10% людей имеют нечеткое представление о цветах и поэтому могут неправильно интерпретировать значение. У разных групп людей различное восприятие цветов; кроме того, в разных профессиях существуют свои соглашения о значении отдельных цветов. Пользователи на основании полученных знаний могут неадекватно интерпретировать один и тот же цвет. Например, водителем красный цвет воспринимается как *опасность*. А у химика красный цвет означает *горячий*.

При использовании слишком ярких цветов или слишком большого их количества отображения становятся путанными. Многообразие цветов сбивает с толку пользователя (так, например, на некоторые абстрактные картины нельзя смотреть длительное время без напряжения) и вызывает у него зрительное утомление. Непоследовательное использование цветов также дезориентирует пользователя.

15.4. Средства поддержки пользователя

В первом разделе этой главы был предложен принцип проектирования, согласно которому интерфейс пользователя должен всегда обеспечивать некоторый тип оперативной справочной системы. Справочные системы – один из основных аспектов проектирования интерфейса пользователя. Справочную систему приложения составляют:

- сообщения, генерируемые системой в ответ на действия пользователя;
- диалоговая справочная система;
- документация, поставляемая с системой.

Таблица 15.4. Факторы проектирования текстовых сообщений

Фактор	Описание
Содержание	Справочная система должна знать, что делает пользователь, и реагировать на его действия сообщениями соответствующего содержания
Опыт пользователя	Если пользователи хорошо знакомы с системой, им не нужны длинные и подробные сообщения. В то же время начинающим пользователям такие сообщения покажутся сложными, малопонятными и слишком краткими. В справочной системе должны поддерживаться оба типа сообщений, а также должны быть средства, позволяющие пользователю управлять сложностью сообщений

Окончание табл. 15.4

Фактор	Описание
Профессиональный уровень пользователя	Сообщения должны содержать сведения, соответствующие профессиональному уровню пользователей. В сообщениях для пользователей разного уровня необходимо применять разную терминологию
Стиль сообщений	Сообщения должны иметь положительный, а не отрицательный оттенок. Всегда следует использовать активный, а не пассивный тон обращения. В сообщениях не должно быть оскорблений или попыток пошутить
Культура	Разработчик сообщений должен быть знаком с культурой той страны, где продается система. Сообщение, вполне уместное в культуре одной страны, может оказаться неприемлемым в другой

Поскольку проектирование полезной и содержательной информации для пользователя — дело весьма серьезное, оно должно оцениваться на том же уровне, что и архитектура системы или программный код. Проектирование сообщений требует значительного времени и немалых усилий. Уместно привлекать к этому процессу профессиональных писателей и художников-графиков. При проектировании сообщений об ошибках или текстовой справки необходимо учитывать факторы, перечисленные в табл. 15.4.

15.4.1. Сообщения об ошибках

Первое впечатление, которое пользователь получает при работе с программной системой, основывается на сообщениях об ошибках. Неопытные пользователи, совершив ошибку, должны понять появившееся сообщение об ошибке.

Новички и опытные пользователи должны предвидеть ситуации, при которых могут возникнуть сообщения об ошибках. Например, пусть пользователем системы является медсестра госпиталя, работающая в отделении интенсивной терапии. Обследование пациентов выполняется на соответствующем оборудовании, связанном с вычислительной системой. Чтобы просмотреть текущее состояние пациента, пользователь системы выбирает пункт меню Показать и набирает имя пациента в поле ввода (рис. 15.9).

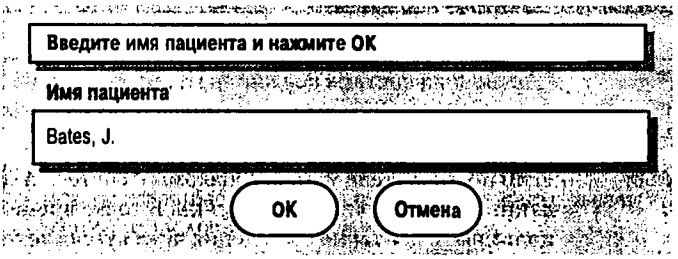


Рис. 15.9. Ввод имени пациента

Пусть медсестра ввела имя пациента Bates, вместо Pates. Система не находит пациента с таким именем и генерирует сообщение об ошибке. Сообщения об ошибке

должны быть всегда вежливыми, краткими, последовательными и конструктивными, не содержать оскорблений. Не следует также использовать звуковые сигналы или другие звуки, которые могут сбить с толку пользователя. Неплохо включить в сообщения варианты исправления ошибки. Сообщение об ошибке должно быть связано с контекстно-зависимой справкой.

На рис. 15.10 показаны примеры двух сообщений об ошибке. Сообщение, расположенное слева, спроектировано плохо. Оно негативно (обвиняет пользователя в совершении ошибки), не адаптировано к уровню знаний и опытности пользователя, не учитывает содержания ошибки. В этом сообщении не предлагаются способы исправления сложившейся ситуации. Кроме того, в сообщении использованы специфические термины (номер ошибки), не понятные пользователю. Сообщение справа гораздо лучше. Оно положительно, в нем используются медицинские термины и предлагается простой способ исправления ошибки посредством щелчка на одной из кнопок. В случае необходимости пользователь может вызвать справку.

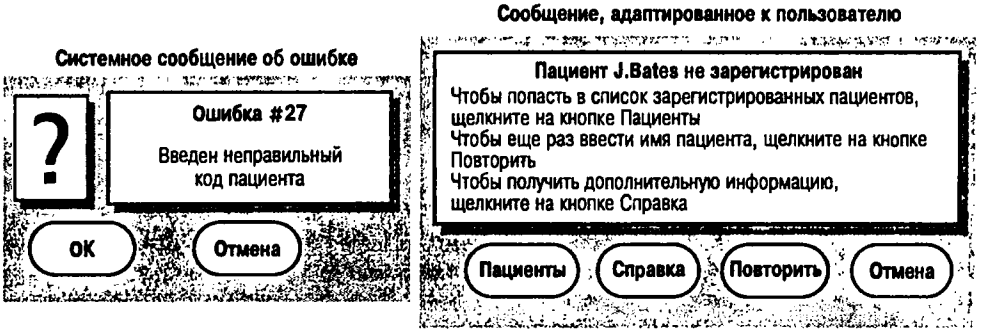


Рис. 15.10. Примеры сообщений об ошибках

15.4.2. Проектирование справочной системы

При получении сообщения об ошибке пользователь часто не знает, что делать, и обращается к справочной системе за информацией. Справочная система должна предоставлять разные типы информации: как ту, что помогает пользователю в затруднительных ситуациях, так и конкретную информацию, которую ищет пользователь. Для этого справочная система должна иметь разные средства и разные структуры сообщений.

Справочная система должна обеспечивать пользователю несколько различных точек входа (рис. 15.11). Пользователь может войти в нее на верхнем уровне ее иерархической структуры и здесь обозреть все разделы справочной информации. Другие точки входа в справочную систему – с помощью окон сообщений об ошибках или путем получения описания конкретной команды приложения.

Все справочные системы имеют сложную сетевую структуру, в которой каждый раздел справочной информации может ссылаться на несколько других информационных разделов. Структура такой сети, как правило, иерархическая, с перекрестными ссылками, как показано на рис. 15.11. Наверху структурной иерархии содержится общая информация, внизу – более подробная.

При использовании справочной системы возникают проблемы, связанные с тем, что пользователи входят в сеть после совершения ошибки и затем перемещаются по сети справочной системы. Через некоторое время они запутываются и не могут определить, в

каком месте справочной системы находятся. В такой ситуации пользователи должны завершить сеанс работы со справочной системой и вновь начать работу с некоторой известной точки справочной сети.

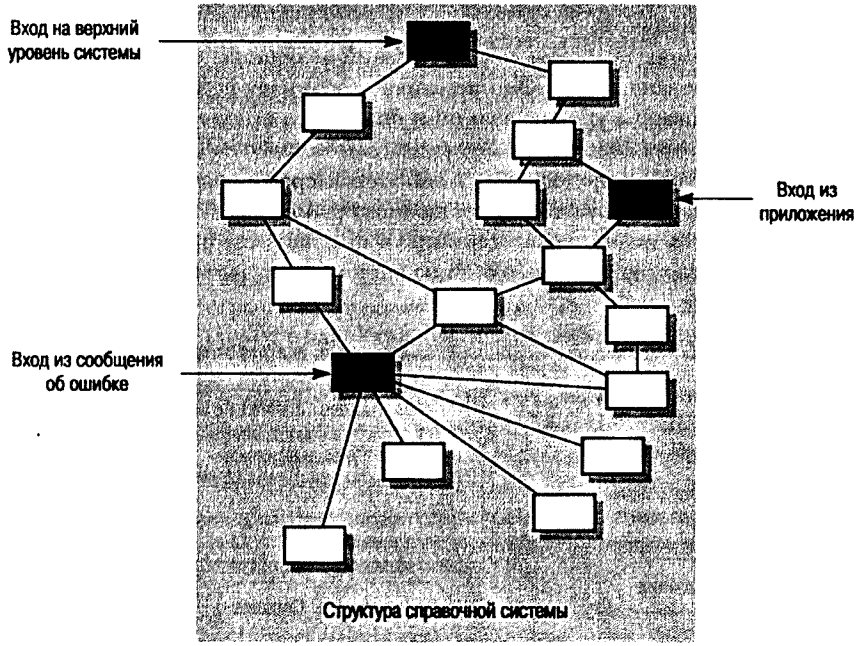


Рис. 15.11. Точки входа в справочную систему

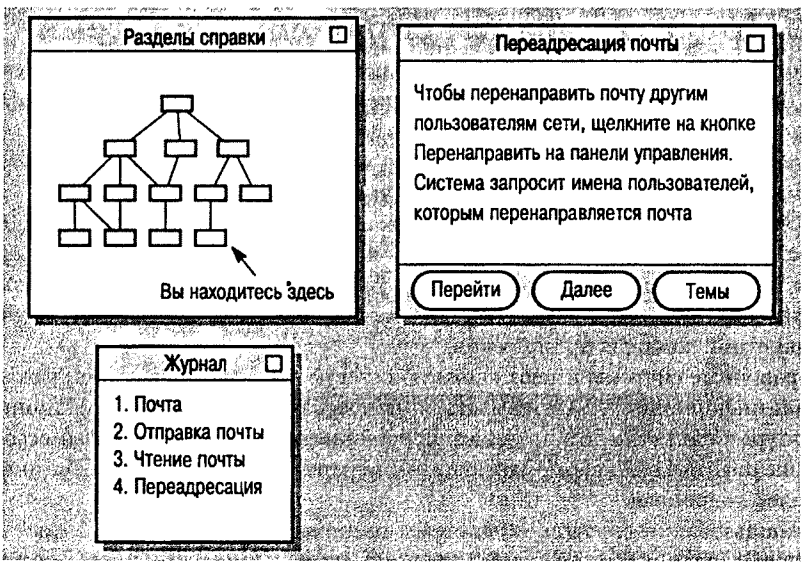


Рис. 15.12. Окна справочной системы

Отображение справочной информации в нескольких окнах упрощает подобную ситуацию. На рис. 15.12 показан экран, на котором расположены три окна справки. Однако пространство экрана всегда ограничено и разработчику следует помнить, что дополнительные окна могут скрыть другую нужную информацию.

Тексты справочной системы необходимо готовить совместно с разработчиками приложения. Справочные разделы не должны быть просто воспроизведением руководства пользователя, поскольку информация на бумаге и на экране воспринимается по-разному. Сам текст (а также его расположение и стиль) должен быть тщательно продуман, чтобы его можно было прочитать в окне относительно малого размера. Раздел справки Переадресация почты на рис. 15.12 сравнительно небольшой – в любом справочном разделе должна быть только самая необходимая информация. В окне, отображающем справочный раздел, расположены три кнопки: для показа дополнительной информации, для перемещения по тексту раздела и для вызова списка справочных тем.

В окне Журнал[1] показан список уже просмотренных разделов. Можно вернуться в один из них, выбрав соответствующий пункт из списка. Окно навигации по справочной системе – это графическая "карта" сети справочной системы. Текущая позиция на карте должна быть выделена цветом, тенями или, как в нашем случае, подписью.

У пользователей есть несколько возможностей перемещения между разделами справочной системы: можно перейти к разделу непосредственно из отображаемого раздела, можно выбрать нужный раздел из окна Журнал, чтобы просмотреть его еще раз, и, наконец, можно выбрать соответствующий узел на карте справочной сети и перейти к этому узлу.

Справочную систему можно реализовать в виде группы связанных Web-страниц или с помощью обобщенной гипертекстовой системы, интегрированной с приложением. Иерархическая структура легко реализуется в виде гипертекстовых ссылок. Web-системы имеют преимущества: они просты в реализации и не требуют специального программного обеспечения. Однако при создании контекстно-зависимой справки могут возникнуть трудности при связывании ее с приложением.

15.4.3. Документация пользователя

Строго говоря, документация не является частью пользовательского интерфейса, однако проектирование оперативной справочной поддержки вместе с документацией является хорошим правилом. Системные руководства предоставляют более подробную информацию, чем диалоговые справочные системы, и строятся так, чтобы быть полезными пользователям разного уровня.

Для того чтобы документация, поставляемая совместно с программной системой, была полезна всем системным пользователям, она должна содержать пять описанных ниже документов (или, может быть, глав в одном документе) (рис. 15.13).

1. *Функциональное описание*, в котором кратко представлены функциональные возможности системы. Прочитав функциональное описание и вводное руководство, пользователь должен определить, та ли это система, которая ему нужна.

---

[1] *Такое окно в англоязычных программных приложениях обычно называется History (История).* – Прим. ред.

2. *Документ по инсталляции системы*, в котором содержится информация по установке системы. Здесь должны быть сведения о дисках, на которых поставляется система, описание файлов, находящихся на этих дисках, и минимальные требования к конфигурации. В документе должна быть инструкция по инсталляции и более подробная информация по установке файлов, зависящих от конфигурации системы.

3. *Вводное руководство*, представляющее неформальное введение в систему, описывающее ее "повседневное" использование. В этом документе должна содержаться информация о том, как начать работу с системой, как использовать общие возможности системы. Все описания должны быть снабжены примерами и содержать сведения о том, как восстановить систему после ошибки и как начать заново работу. В книге [68] предложен эффективный способ составления вводного руководства, при котором основное внимание уделено восстановлению системы после ошибок, а вся остальная информация, необходимая пользователям, сводится к минимуму.

4. *Справочное руководство*, в котором описаны возможности системы и их использование, представлен список сообщений об ошибках и возможные причины их появления, рассмотрены способы восстановления системы после выявления ошибок.

5. *Руководство администратора*, необходимое для некоторых типов программных систем. В нем дано описание сообщений, генерируемых системой при взаимодействии с другими системами, и описаны способы реагирования на эти сообщения. Если в систему включена аппаратная часть, то в руководстве администратора должна быть информация о том, как выявить и устранить неисправности, связанные с аппаратурой, как подключить новые периферийные устройства и т.п.

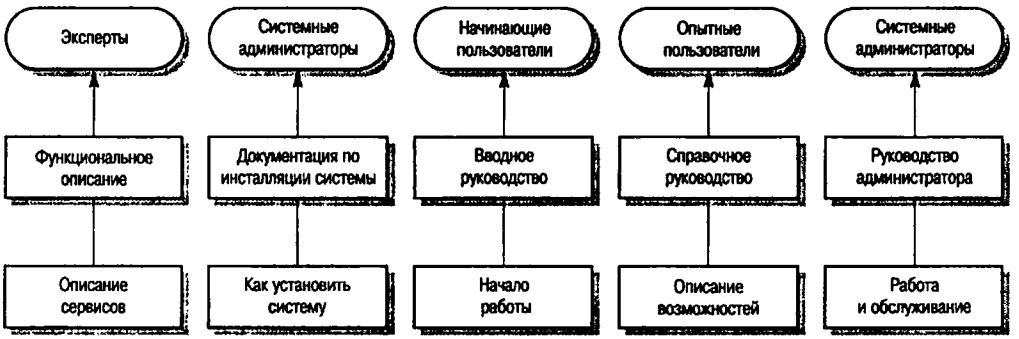


Рис. 15.13. Типы пользовательской документации

Вместе с перечисленными руководствами необходимо предоставлять другую удобную в работе документацию. Для опытных пользователей системы удобны разного вида предметные указатели, которые помогают быстро просмотреть список возможностей системы и способы их использования.

15.5. Оценивание интерфейса

Это процесс, в котором оценивается удобство использования интерфейса и степень его соответствия требованиям пользователя. Таким образом, оценивание интерфейса является частью общего процесса тестирования и аттестации систем ПО.

В идеале оценивание должно проводиться в соответствии с показателями удобства использования интерфейса, перечисленными в табл. 15.5. Каждый из этих показателей можно оценить численно. Например, изучаемость можно оценить следующим образом: опытный оператор после трехчасового обучения должен уметь использовать 80% функциональных возможностей системы. Однако чаще удобство использования интерфейса оценивается качественно, а не через числовые показатели.

Таблица 15.5. Показатели удобства использования интерфейса

Показатель	Описание
Изучаемость	Количество времени обучения, необходимое для начала продуктивной работы с системой
Скорость работы	Скорость реакции системы иа действия пользователя
Устойчивость	Устойчивость системы к ошибкам пользователя
Восстанавливаемость	Способность системы восстанавливаться после ошибок пользователя
Адаптируемость	Способность системы "подстраиваться" к разным стилям работы пользователей

Полное оценивание пользовательского интерфейса может оказаться весьма дорогостоящим, в этот процесс будут вовлечены специалисты по когнитивной психологии и дизайнеры. В процесс оценивания могут входить разработка и выполнение ряда статистических экспериментов с пользователями в специально созданных лабораториях и с необходимым для наблюдения оборудованием. Такое оценивание интерфейса экономически нерентабельно для систем, разрабатываемых в небольших организациях с ограниченными ресурсами.

Существуют более простые и менее дорогостоящие методики оценивания интерфейсов пользователя, позволяющие выявить отдельные дефекты в интерфейсах.

1. Анкеты, в которых пользователь дает оценку интерфейсу.
2. Наблюдения за работой пользователей с последующим обсуждением их способов использования системы при решении конкретных задач.
3. Видеонаблюдения типичного использования системы.
4. Добавление в систему программного кода, который собирал бы информацию о наиболее часто используемых системных сервисах и наиболее распространенных ошибках.

Анкетирование пользователей – относительно дешевый способ оценки интерфейса. Вопросы должны быть точными, а не общими. Не следует использовать вопросы типа "Пожалуйста, прокомментируйте практичность системы", так как ответы, вероятно, будут существенно различаться. Лучше задавать конкретные вопросы, напри-

мер: "Оцените понятность сообщений об ошибках по шкале от 1 до 5. Оценка 1 означает полностью понятное сообщение, 5 – малопонятное". На такие вопросы легче ответить и более вероятно получить в результате полезную для улучшения интерфейса информацию.

Во время заполнения анкеты пользователи должны обязательно оценить собственный опыт и знания. Такого рода сведения позволят разработчикам зафиксировать, пользователи с каким уровнем знаний имеют проблемы с интерфейсом. Если проект интерфейса уже создан и прошел оценивание в бумажном виде, анкеты можно использовать даже до полной реализации системы.

При наблюдении пользователей за работой оценивается, как они взаимодействуют с системой, какие используют сервисы, какие совершают ошибки и т.п. Вместе с наблюдениями могут проводиться семинары, на которых пользователи рассказывают о своих попытках решить те или иные проблемы и о том, как они понимают систему и как используют ее для достижения целей.

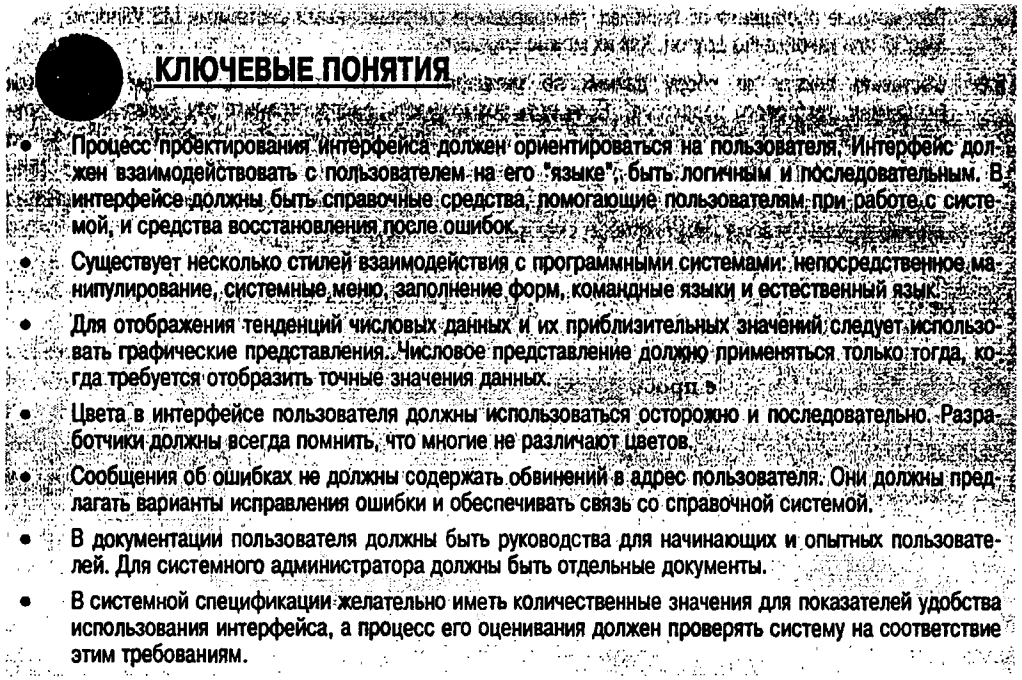
Видеооборудование относительно недорого, поэтому к непосредственному наблюдению можно добавить видеозапись пользовательских семинаров для последующего анализа. Полный анализ видеоматериалов дорогостоящий и требует специально оснащенного комплекта с несколькими камерами, направленными на пользователя и на экран. Однако видеозапись отдельных действий пользователя может оказаться полезной для обнаружения проблем. Чтобы определить, какие именно действия вызывают проблемы у пользователя, следует прибегнуть к другим методам оценивания.

Анализ видеозаписей позволяет разработчику установить, много ли движений руками вынужден совершать пользователь (в некоторых системах пользователю постоянно приходится переходить с клавиатуры на мышь), и обнаружить неестественные движения глаз. Если при работе с интерфейсом требуется часто смещать зрительный фокус, пользователь может совершить больше ошибок и пропустить какие-либо части изображения.

Вставка в программу кода, собирающего статистические данные при использовании системы, улучшает интерфейс несколькими способами. Обнаруживаются наиболее часто используемые операции. Интерфейс изменяется так, чтобы эти операции выбирались более быстро по сравнению с другими. Например, в вертикальном или выпадающем меню наиболее часто используемые команды должны находиться вверху списка. Такой код также позволит обнаружить и изменить команды, способствующие появлению ошибок.

Наконец, в каждой программе должны быть несложные средства, с помощью которых пользователь сможет передавать разработчикам сообщения с "жалобами". Такие средства убеждают пользователей в том, что с их мнением считаются. А разработчики интерфейса и другие специалисты могут получить быструю обратную связь относительно отдельных проблем интерфейса.

Ни один из этих далеко не сложных методов оценки пользовательского интерфейса не является надежным и не гарантирует решения всех проблем интерфейса. Вместе с тем перед выпуском системы эти методы можно применить в группе добровольцев, не затрачивая значительных средств. При этом обнаруживается и исправляется большинство проблем в интерфейсе пользователя.



КЛЮЧЕВЫЕ ПОНЯТИЯ

- Процесс проектирования интерфейса должен ориентироваться на пользователя. Интерфейс должен взаимодействовать с пользователем на его "языке", быть логичным и последовательным. В интерфейсе должны быть справочные средства, помогающие пользователям при работе с системой, и средства восстановления после ошибок.
- Существует несколько стилей взаимодействия с программными системами: непосредственное манипулирование, системные меню, заполнение форм, командные языки и естественный язык.
- Для отображения тенденций числовых данных и их приблизительных значений следует использовать графические представления. Числовое представление должно применяться только тогда, когда требуется отобразить точные значения данных.
- Цвета в интерфейсе пользователя должны использоваться осторожно и последовательно. Разработчики должны всегда помнить, что многие не различают цветов.
- Сообщения об ошибках не должны содержать обвинений в адрес пользователя. Они должны предлагать варианты исправления ошибки и обеспечивать связь со справочной системой.
- В документации пользователя должны быть руководства для начинающих и опытных пользователей. Для системного администратора должны быть отдельные документы.
- В системной спецификации желательно иметь количественные значения для показателей удобства использования интерфейса, а процесс его оценивания должен проверять систему на соответствие этим требованиям.

Упражнения

15.1. В разделе 15.1 отмечалось, что объекты, которыми манипулирует пользователь, должны отображать его понятия предметной области приложения ПО (а не компьютерной предметной области). Предложите подходящие объекты манипулирования для следующих типов пользователей и систем.

- Автоматизированный каталог товаров для ассистента на складе.
- Система наблюдения за безопасностью самолета для летчика гражданской авиации.
- Финансовая база данных для менеджера.
- Система управления патрульными машинами для полицейского.

15.2. Опишите ситуации, в которых неразумно или невозможно поддерживать интерфейс пользователя.

15.3. Какие факторы следует учитывать при проектировании интерфейсов, использующих меню, для таких систем, как банкоматы? Опишите основные черты интерфейса банкоматв, которым вы пользуетесь.

15.4. Предложите способы адаптации пользовательского интерфейса в системах электронной коммерции (например, виртуального книжного магазина или магазина музыкальных дисков) для пользователей, имеющих физические недостатки, например плохое зрение или проблемы опорно-двигательной системы.

15.5. Обсудите преимущества графического способа отображения информации и приведите четыре примера приложений, в которых более уместно использовать графическое представление числовых данных, а не табличное.

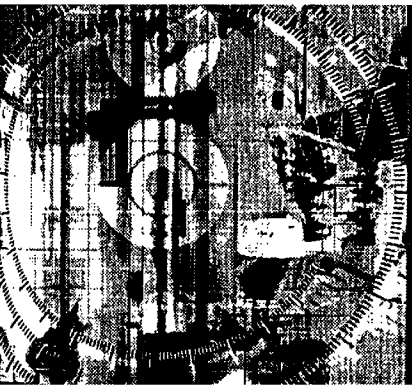
15.6. Какими основными принципами следует руководствоваться при использовании цветов в интерфейсах пользователя? Предложите более эффективный способ использования цветов в интерфейсе любого известного вам приложения.

15.7. Рассмотрите сообщения об ошибках, генерируемые операционными системами MS Windows, Unix, MacOS или какой-либо другой. Как их можно улучшить?

15.8. Составьте анкету по сбору данных об интерфейсе какой-либо известной вам программы (например, текстового редактора). Если есть возможность, распространите эту анкету среди других пользователей и попытайтесь оценить результаты анкетирования. Что вы узнали об интерфейсе программы из анкет?

15.9. Обсудите, этично ли разрабатывать программные системы, не согласовав с конечными пользователями те элементы системы, которые они будут контролировать.

15.10. С какими этическими проблемами сталкиваются разработчики интерфейсов, когда пытаются согласовать запросы конечных пользователей системы с требованиями организации, которая оплачивает разработку данной системы?



Критические системы

16 Надежность систем

Цели

Цель настоящей главы – дать понятие функциональной надежности и ее важности для критических систем. Прочитав эту главу, вы должны:

- знать четыре составляющие функциональной надежности: работоспособность, безотказность, безопасность и защищенность;
- понимать, что критическими являются такие системы, при неправильном функционировании которых могут возникать тяжелые человеческие или экономические потери;
- знать основные способы достижения функциональной надежности системы: сведение к минимуму ошибок во время разработки, выявление и устранение ошибок во время эксплуатации и уменьшение последствий неправильного функционирования системы.

Содержание

Известно, что компьютерные системы имеют недостатки, т.е. без явных причин иногда выходят из строя, и не всегда ясно, что требуется для восстановления их работоспособности. Программы, выполняемые на таких компьютерах, работают неверно, порой искажая данные. Необходимо учиться жить с этими недостатками, не теряя веры в то, что существуют персональные компьютеры, которые обычно работают нормально.

Функциональную надежность компьютерных систем можно определить степенью доверия к ним, т.е. уверенностью, что система будет работать так, как предполагается, и что сбоев не будет. Это свойство нельзя оценить количественно. Для этого используются такие относительные термины, как "ненадежные", "очень надежные" или "сверхнадежные", отражающие различную степень доверия к системе.

Надежность и полезность — это, конечно, разные вещи. Программа текстового редактора, которую я использовал при написании книги, является не очень надежной, но весьма полезной системой. Зная это, я часто сохранял работу, многократно ее копируя. Этими действиями я компенсировал недостатки системы, снижая риск потери информации в случае ее отказа.

Существует четыре осиовные составляющие функциональной надежности программных систем (рис. 16.1), неформальные определения которых приведены ниже.

1. *Работоспособность* – свойство системы выполнять свои функции в любое время эксплуатации.
2. *Безотказность* – свойство системы корректно (так, как ожидает пользователь) работать весь заданный период эксплуатации.
3. *Безопасность* — свойство системы, гарантирующее, что она безопасна для людей и окружающей среды.
4. *Защищенность* — свойство системы противостоять случайным или намеренным вторжениям в нее.

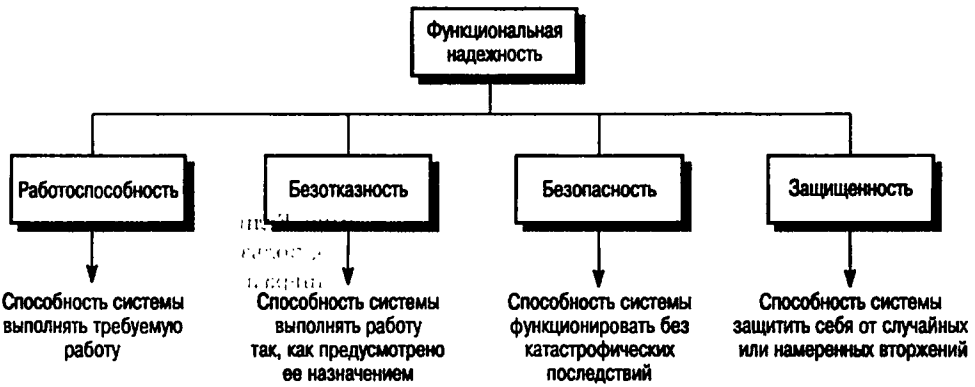


Рис. 16.1. Составляющие надежности системы

Работоспособность и безотказность систем обсуждаются в главе 17. Эти показатели носят вероятностный характер и могут быть выражены количественно. Безопасность и защищенность редко выражаются в виде числовых показателей, ио их можно сравнивать по относительной шкале уровней. Например, безопасность уровня 1 меньше безопасности уровня 2, которая, в свою очередь, меньше безопасности уровня 3, и т.д.

Дополнительные меры, повышающие функциональную надежность системы, могут резко увеличивать стоимость ее разработки. На рис. 16.2 показана зависимость между стоимостью разработки и различными уровнями функциональной надежности. Здесь подразумевается, что функциональная надежиость содержит все составляющие: работоспособность, безотказность, безопасность и защищенность. Экспоненциальный характер зависимости "стоимость–надежность" не позволяет говорить о возможности создания систем со стопроцентной надежностью, так как стоимость их создания была бы очень большой.

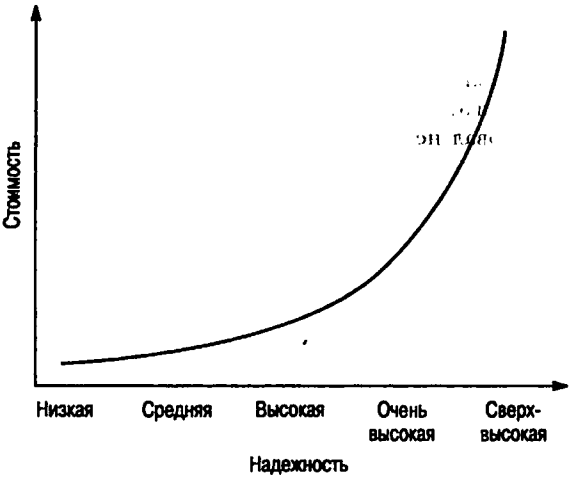


Рис. 16.2. Зависимость между стоимостью разработки системы и ее надежностью

Высокие уровни функциональной надежности могут быть достигнуты только за счет уменьшения эффективности работы системы. Например, надежное программное обеспечение предусматривает дополнительные, часто избыточные, коды для проверки нештатных состояний системы. Это усложняет систему и увеличивает объем памяти, необходимый для ее эффективной работы. Но в ряде случаев надежность более важна, чем эффективность системы.

1. *Ненадежные системы часто остаются невостребованными.* Если к системе нет доверия пользователя, она не будет востребована. Более того, пользователи могут отказаться от других программных продуктов той же компании-разработчика, поскольку будут также считать их ненадежными.
2. *Стоимость отказа системы может быть огромна.* Для некоторых приложений, таких, как системы управления реакторами или системы навигации, стоимость последствий отказа может превышать стоимость самой системы.
3. *Трудно модернизировать ненадежную систему для повышения ее надежности.* Обычно есть возможность улучшить неэффективную систему, так как в этом случае основные усилия будут затрачены на модернизацию отдельных программных модулей. Систему, к которой нет доверия, трудно улучшить, поскольку ненадежность "распределена" по всей системе.
4. *Существуют возможности компенсировать недостаточную эффективность системы.* Если программная система работает неэффективно, то это постоянный фактор, к кото-

рому пользователь может приспособиться, построив свою работу с его учетом. Ненадежность системы, как правило, проявляется внезапно. Ненадежное программное обеспечение может нарушить работу всей системы, в которую оно интегрировано, и разрушить данные пользователя без предупреждения, что может иметь серьезные последствия.

5. *Ненадежные системы могут быть причиной потери информации.* Сбор и хранение данных — дорогостоящая процедура, часто данные стоят больше, чем компьютерная система, на которой они обрабатываются. Дублирование данных для предотвращения их потери вследствие ненадежности системы потребует значительных усилий и финансовых средств.

Надежность системы зависит от технологии разработки ПО. Многократные тестирования с целью исключения ошибок способствуют разработке надежных систем. Однако нет простой связи между качеством процесса создания и качеством готовой системы. Вопросы взаимосвязи между процессом создания ПО и качеством готовой продукции рассматриваются в главах 24 и 25, где представлены темы управления качеством и модернизации систем.

16.1. Критические системы

Обычно отказ систем, управляемых с помощью ПО, вызывает неудобства, но они не приводят к длительным последствиям. Однако имеются системы, отказы которых могут приводить к значительным экономическим потерям, физическим повреждениям или создавать угрозу человеческой жизни. Такие системы обычно называют *критическими*. Функциональная надежность — необходимое требование к критическим системам, и все ее составляющие (работоспособность, безотказность, безопасность и защищенность) очень важны. Не менее важен для критических систем и высокий уровень надежности.

Существует три основных типа критических систем.

1. *Системы, критические по обеспечению безопасности.* Системы, отказ которых приводит к разрушениям, создает угрозу жизни человека или наносит вред окружающей среде. В качестве примера можно привести систему управления производством на химическом заводе.
2. *Системы, критические для целевого назначения.* Системы, отказ которых может привести к ошибкам в действиях, направленных на обеспечение определенной цели. Примером может служить навигационная система космического корабля.
3. *Системы, критические для бизнеса.* Отказ таких систем может нанести вред делу, в котором они используется. Примером является система, обслуживающая счета клиентов в банке.

Цена ошибки критической системы часто очень велика. Она включает прямые расходы, связанные с внесением изменений в систему или ее заменой, косвенные расходы, например судебные, и расходы, связанные с потерями в бизнесе. Из высокой возможной цены отказа системы следует, что качество методов разработки и сам процесс создания ПО обычно более важны, чем стоимость применения этих методов.

Поэтому при создании критических систем обычно используются испытанные методы разработки, а не новые, еще не имевшие большого практического применения. Только сравнительно недавно такие относительно новые методы, как, например, объектно-ориентированные, стали использоваться для разработки критических систем, вместе с

тем до сих пор при разработке многих критических систем все еще применяются функционально-ориентированные методы.

С другой стороны, методы разработки ПО, которые обычно нерентабельны, могут использоваться для разработки критических систем, например метод формальных спецификаций и формализованной проверки программ на соответствие таким спецификациям. Одной из причин использования этих методов является уменьшение количества требующегося тестирования. Для критических систем стоимость проверки и аттестации обычно очень высока и может составлять более 50% общей стоимости системы.

Хотя эта книга посвящена разработке программных систем, а не общей теории систем, необходимо отметить, что функциональная надежность — общесистемное понятие. Рассматривая надежность критических систем, можно выделить три типа системных "компонентов", склонных к отказу.

1. Аппаратные средства системы, отказывающие либо из-за ошибок конструирования, лнбо из-за ошибок изготовления, либо из-за полного износа.
2. Программное обеспечение системы, которое может отказывать из-за ошибок либо в технических требованиях к системе, либо в архитектуре системы, либо в программном коде.
3. Человеческий фактор, который своими действиями нарушает правильную работу системы.

Таким образом, если цель состоит в том, чтобы повысить надежность системы, необходимо рассматривать все эти аспекты во взаимосвязи. Я поясняю это положение на нескольких примерах в других главах этой части книги.

16.1.1. Системы, критические по обеспечению безопасности

В этом разделе в качестве примера системы, критической по обеспечению безопасности, рассмотрим систему, управляющую дозировкой инъекций инсулина при заболевании диабетом. Предполагается, что большинство читателей имеют общее представление об этом заболевании и его лечении. Эта система описана в главе 9, где была представлена ее формальная спецификация.

Диабет — заболевание, при котором человеческий организм не может выработать достаточное количество гормона, называемого инсулином и регулирующего содержание сахара в крови. Если уровень инсулина в организме будет избыточным, содержание сахара в крови может понизиться, что влечет за собой очень серьезные последствия — прекращение питания мозга, приводящее к потере сознания и даже летальному исходу. Если же организм вырабатывает инсулин в недостаточном количестве, то уровень сахара повышается, что приводит к нарушению зрения, заболеванию почек и всего организма.

Благодаря появлению миниатюрных датчиков стало возможным создание автоматизированной системы инъекций инсулина. Она контролирует уровень сахара в крови, и, если необходимо, в организм вводится соответствующая доза инсулина. Конечно, такие системы могут пока работать только в стационарных условиях. В дальнейшем, будучи подключеиными к человеку, они станут доступны широкому кругу больных диабетом.

Для работы такой системы микродатчик вживляется в тело больного. Этот датчик контролирует определенный параметр крови, который характеризует уровень сахара. Результат посылается в контрольный блок, который вычисляет уровень сахара, определяет необходимую дозу инсулина и посылает сигнал для инъекций постоянно подсоединенной

игле. Совершенно очевидно, что такой системой должна управлять надежная программа. На рис. 16.3 показаны компоненты и организация системы дозировки инъекций инсулина. На рис. 16.4 показана модель потока данных, где видно, как входное значение уровня сахара в крови преобразуется в последовательность команд управления дозировкой инъекций инсулина.

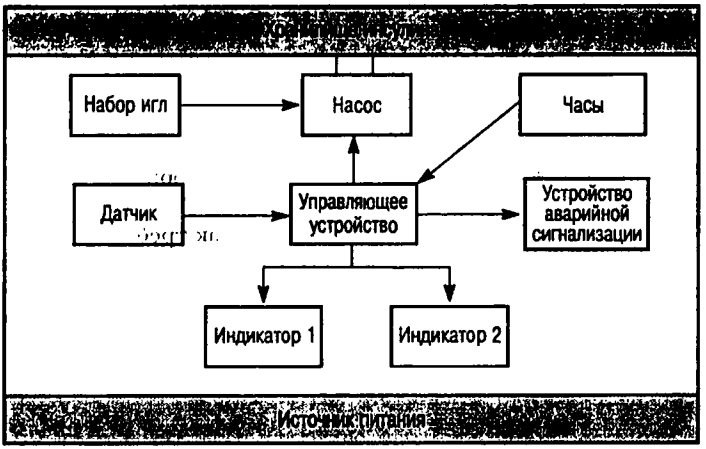


Рис. 16.3. Структура системы дозировки инъекций инсулина

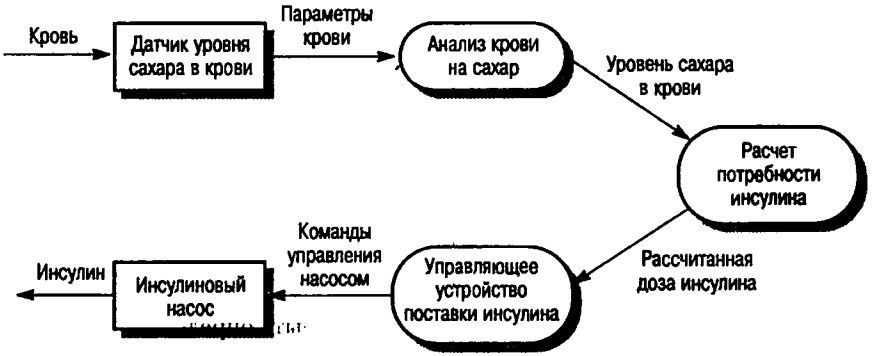


Рис. 16.4. Модель потока данных системы дозировки инъекций инсулина

Надежность системы дозировки инъекций инсулина характеризуется тремя составляющими.

1. *Работоспособность.* При возникновении критической ситуации система должна быть готовой ввести необходимую дозу инсулина.
2. *Безотказность.* В зависимости от уровня сахара в крови система должна правильно определить дозу инсулина и ввестн ее пациенту.
3. *Безопасность.* Отказ такой системы может привести к чрезмерной дозе инсулина, что представляет опасность для жизни больного. Необходимо добиться, чтобы такого рода отказов в системе не было.

В последующих главах разъясняется, как эти составляющие можно точно определить и проверить.

16.2. Работоспособность и безотказность

В этом разделе обсуждаются две близко связанные составляющие функциональной надежности — работоспособность и безотказность. Под работоспособностью системы подразумевается способность предоставлять пользователю все необходимые системные сервисы по мере возникновения потребности в них. Безотказность — это способность системы предоставлять именно те сервисы, которые заложены в систему ее спецификацией. Из этого следует, что безотказность более общий показатель, включающий в себя свойство работоспособности, поскольку если система не работоспособна, то о безотказности вообще речи идти не может.

Однако необходимо и различать эти свойства, так как требования к работоспособности и безотказности в разных системах могут быть различны. Например, некоторые системы могут иметь сравнительно частые сбои, но они также могут быстро восстанавливаться после сбоев. У таких систем сравнительно низкие требования к безотказности. В то же время они могут иметь высокие требования к работоспособности в связи с необходимостью непрерывного обслуживания пользователей.

Наглядный пример такой системы — коммутатор телефонной станции. Снимая трубку телефона, пользователь слышит зуммер, означающий, что "линия свободна" и система готова выполнить требования пользователя. В случае сбоя такая система должна быть легко восстанавливаема. Коммутатор телефонной станции имеет средства для восстановления неправильной работы и разрешает повторную попытку соединения. Это выполняется очень быстро, и абонент телефона может даже не почувствовать, что был сбой. В такой ситуации основным требованием к функциональной надежности системы является безотказность.

Другое различие между этими показателями заключается в том, что работоспособность системы зависит также от времени, которое требуется для устранения неисправности. Так как, если система А отказывает один раз в год, а система В отказывает раз в месяц, то система А более надежна, чем В. Однако, если системе А необходимо три дня для восстановления работоспособности, а системе В для этого достаточно всего 10 минут, то безотказность системы В выше, чем системы А. Исходя из этих соображений пользователь скорее выберет систему В, чем А.

Безотказность и работоспособность системы могут быть определены более точно следующим образом.

1. *Безотказность* — это способность системы безотказно работать определенное время с указанной целью в определенном окружении.
2. *Работоспособность* — это способность системы правильно функционировать и предоставлять вовремя требуемые сервисы.

Одна из практических проблем, возникающая при разработке систем, заключается в том, что наши интуитивные понятия безотказности и работоспособности часто оказываются шире приведенных выше формулировок. При рассмотрении функциональной надежности системы должны быть приняты во внимание окружение, в котором будет действовать система, и цель, для которой она используется. Следовательно, оценка безотказности в одном окружении не обязательно переносится в другое окружение, где система используется по-другому.

Для примера рассмотрим безотказность системы программного обеспечения в двух средах (окружении) — в офисе и в университете. В офисе пользователи строго следуют инструкциям по работе с системой и не имеют времени и возможностей экспериментировать с ней. В университетской среде студенты пробуют все возможности системы и часто непредвиденными способами, что может привести к отказам системы, которые никогда бы не встретились в офисной среде.

Также важны способы использования системы и реакция человека на ее работу. Представим себе, что у автомобиля неисправна стеклоочистительная система — происходит сбой в работе стеклоочистителей. В дождливую погоду работа этой системы очень важна. Безотказность такой системы будет определяться местностью, где происходит действие, и реакцией водителя. Для водителя из Сиэтла (влажный климат) этот отказ будет более чувствителен, чем для водителя из Лас-Вегаса (сухой климат). Водитель из Сиэтла будет считать, что система не является безотказной, в то время как водитель из Лас-Вегаса вообще никогда не столкнется с этой проблемой.

В определении работоспособности и безотказности системы не рассматривается тяжесть и последствия отказов системы. Людям особенно небезразличны отказы, которые имеют серьезные последствия; от этого зависит восприятие безотказности системы. Например, работа автомобильного двигателя, который дает сбои сразу после запуска, а затем после перезапуска работает исправно, раздражает. Но это не затрагивает нормального режима эксплуатации автомобиля.

Считается, что система ведет себя надежно, если ее работа соответствует заданному алгоритму. Однако возможны ситуации, когда работа системы не совсем соответствует ожиданиям пользователей. К сожалению, системные требования часто бывают или неполны или некорректны. Разработчики в таких случаях должны сами решать, как будет вести себя система, но, не всегда являясь специалистами в конкретной области применения системы, они не могут учесть все факторы и запрограммировать такое поведение системы, которое необходимо пользователю.

Как показывает опыт, наиболее важными составляющими функциональной надежности являются безотказность и работоспособность. Если же система ненадежна, то трудно гарантировать ее безопасность или защищенность. Ненадежность системы влечет за собой большие материальные потери, такие системы приобретают репутацию некачественных и в дальнейшем теряют доверие потребителей.

Безотказность системы определяется отсутствием сбоев. Отказы системы могут происходить из-за плохого или неправильного ее обслуживания, могут быть следствием ошибок в алгоритме, а могут быть вызваны неисправностями систем связи. Однако во многих случаях причиной ошибочного поведения системы являются дефекты в самой системе. При рассмотрении безотказности полезно понимать различие в терминах *сбой*, *ошибка* и *отказ*, которые определены в табл. 16.1.

Таблица 16.1. Терминология безотказности

Термин	Описание
Отказ системы	Прекращение функционирования системы
Системные ошибки	Ошибочное поведение системы, не соответствующее ее спецификации
Сбой системы	Неправильное поведение системы, непредвиденное ее разработчиками
Ошибка оператора	Неверные действия пользователя, вызвавшие сбой в работе системы

Сбои не обязательно приводят к отказам системы, поскольку они могут быть кратковременными и система может прийти к нормальному функционированию раньше, чем произойдет отказ. Системные ошибки также не обязательно приводят к отказам системы, так как системы имеют защиту, гарантирующую, что ошибочный режим будет обнаружен и исправлен.

Терминология, приведенная в табл. 16.1, помогает понять три дополняющих друг друга подхода, используемых для повышения безотказности систем.

1. *Предотвращение сбоев.* Подход к разработке ПО, минимизирующий возможность появления ошибок и/или обнаруживающий ошибки прежде, чем они приведут к сбоям системы. Пример такого подхода – исключение подверженных ошибкам определенных конструкций языков программирования (например, указателей) и постоянный анализ программ для обнаружения различных аномалий программного кода (см. главу 19).
2. *Обнаружение ошибок и их устранение.* Использование разнообразных методов проверки системы в различных режимах позволяет обнаружить ошибки и устранить их до ввода системы в эксплуатацию. Регулярное тестирование системы и ее отладка – пример данного подхода.
3. *Устойчивость к сбоям.* Использоваиие специальных методов, гарантирующих, что ошибки в системе не приведут к сбоям и что сбои не приведут к отказам системы. Пример такого подхода – применение средств самовосстановления системы с использованием дублирования модулей.

Разработка систем, устойчивых к сбоям, описана в главе 18, где также кратко обсуждаются некоторые методы предотвращения сбоев. Методы разработки ПО, предотвращающие сбои, приведены в главе 24. Вопросы обнаружения системных ошибок обсуждаются в главах 19 и 20.

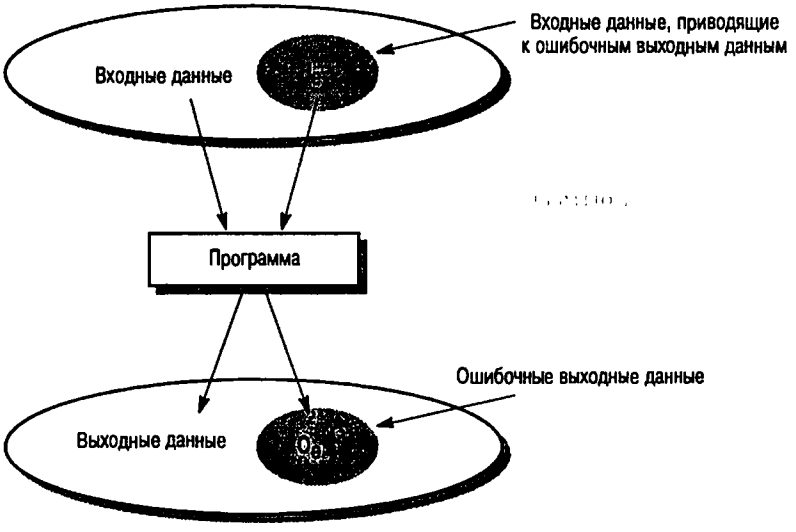


Рис.16.5. Программная система как отображение входных данных на множество выходных данных

Программный сбой приводит к отказу системы, когда "сбойный" программный код выполняется над определенными входными данными, приводящими к сбою системы. При этом программа может корректно работать на других входных данных. На рис. 16.5, заимствованным из работы [221], программная система представлена как отображение множества входных данных (входов) на множество выходных данных (выходов). Программа обрабатывает входные данные, получая выходные данные.

Некоторые из входных данных (см. рис. 16.5) приводят к отказам системы, в результате программа генерирует ошибочные выходные данные. Безотказность программного обеспечения характеризуется вероятностью, с которой при выполнении программы среди множества входных данных встретятся такие, что приведут к ошибочным результатам вычислений.

Существует сложная взаимосвязь между наблюдаемой безотказностью системы и количеством скрытых программных дефектов. В работе [239] подчеркивается, что не все программные ошибки в равной степени вызывают отказ системы. Обычно в множестве входных данных I_e, приводящих к ошибочным выходным данным, имеется ряд данных, вероятность выбора которых больше, чем у других данных. Если эти входные данные не требуют для своей обработки той части ПО, которая содержит ошибки, то системных сбоев не будет. Таким образом, безотказность системы зависит преимущественно от количества входных данных, приводящих к ошибочным результатам во время нормальной эксплуатации системы. Сбои системы, которые проявляются только в исключительных ситуациях, мало влияют на ее надежность.

Надежность системы связана с вероятностью ошибки, проявляющейся во время эксплуатации системы. Устранение программных ошибок в редко используемых системных модулях мало повлияет на повышение безотказности системы. В работе [239] показано, что устранение 60% программных ошибок только на 3% повысит безотказность системы. Это подтверждается и исследованиями ошибок в программных продуктах IBM. В работе [4] показано, что многие ошибки в программных продуктах реально вызывают сбои системы после сотен или тысяч месяцев эксплуатации.

Следовательно, программа может содержать ошибки и все же вызывать доверие пользователей. Сбои программы никогда не будут возникать, если не выбирать входных данных, ведущих к сбоям. Кроме того, опытные пользователи часто работают, зная об ошибках программного обеспечения, вызывающих сбои системы, и умело избегают их. Устранение ошибок в таких случаях не даст практически никакого повышения надежности.

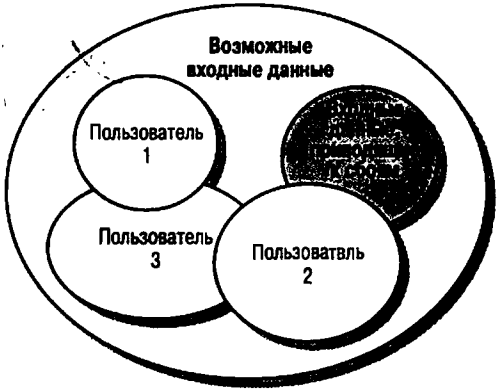


Рис. 16.6. Использование множеств входных данных разными пользователями

Каждый пользователь системы по-своему избегает "встречи" с системными ошибками. Ошибки, с которыми встречается один пользователь, могут никогда не встретиться другому. На рис. 16.6 показано множество входных данных, используемых разными пользователями. Входные данные, выбранные пользователем 2 пересекаются с входными данными, приводящими к сбою системы. Поэтому пользователь 2 будет сталкиваться со сбоями системы. Пользователи 1 и 3 никогда не используют входных данных, приводящих к сбою системы, – для них программное обеспечение всегда будет безотказно.

16.3. Безопасность

Безопасность системы – это свойство, отражающее способность системы функционировать, не угрожая людям или окружающей среде. Там, где безопасность является необходимым атрибутом системы, говорят о системе, критической по обеспечению безопасности. Примерами могут служить контролирующие и управляющие системы в авиации, системы управления процессами на химических и фармацевтических заводах и системы управления автомобилями.

Управление систем, критических по обеспечению безопасности, гораздо проще организовать с помощью аппаратных средств, чем ПО. Однако сейчас строятся системы такой сложности, что управление не может осуществляться только аппаратными средствами. Из-за необходимости управлять большим числом сенсоров и исполнительных механизмов со сложными законами управления требуется управляющее программное обеспечение. В качестве примера можно привести системы управление военными самолетами. Они требуют постоянного программного управления самолетом, гарантирующего безопасность полета.

Программное обеспечение рассматриваемых в этом разделе систем подразделяется на два класса.

1. *Первичное программное обеспечение, критическое по критерию безопасности.* Это ПО включается в систему в виде отдельного блок управления. Неправильная работа такого ПО может быть причиной отказа оборудоваиия, вследствие которого может возникнуть угроза жизни человека или нанесения вреда окружающей среде. Я особо обращаю внимание на этот класс программного обеспечения.
2. *Вторичное программное обеспечение, критическое по критерию безопасности.* Это ПО косвенным образом может привести к непредвиденным последствиям. Примерами могут служить автоматизированные системы в технике, неправильная работа которых может привести к ошибкам в работе объекта и поставить под угрозу жизнь людей. Другой пример такого ПО – медицинская база данных, содержащая описание лекарств, предназначенных пациентам. Ошибки в этой системе могут привести к неправильной дозировке препаратов.

Безотказность и безопасность системы – взаимосвязанные, но, очевидно, различные составляющие функциональной надежности. Конечно, система, критическая по обеспечению безопасности, должна соответствовать своему назначению и функционировать без отказов. Для обеспечения непрерывного функционирования даже в случае ошибок она должна иметь защиту от сбоев. Однако отказоустойчивость не гарантирует безопасности системы. Программное обеспечение может только один раз сработать неправильно, и это приведет к несчастным случаям.

Нельзя быть на сто процентов уверенным, что системное программное обеспечение безопасно и отказоустойчиво. Безотказность ПО не гарантирует его безопасность по ряду причин.

1. В системной спецификации может быть не определено поведение системы в некоторых критических ситуациях. Высокий процент сбоев систем — результат скорее неверных или неполных требований, чем ошибок программирования [49, 109, 226, 252, 8*, 20*, 29*]. После изучения ошибок в программных системах Лутц (Lutz, [226]) делает вывод:

   "...трудности с формулировкой требований к разрабатываемой системе — основная причина ошибок программного обеспечения, связанных с безопасностью, которые нельзя выявить до сборки и тестирования системы".

2. Сбои в работе аппаратных средств могут привести к непредсказуемому поведению системы, в результате чего программное обеспечение сталкивается с непредвиденной ситуацией. Когда системные компоненты близки к состоянию отказа, они могут вести себя неустойчиво и генерировать сигналы, которые могут быть обработаны программным обеспечением непредвиденным образом.

3. Операторы, работающие с системой, могут внести ошибки, которые в особых ситуациях способны привести к сбою системы. Анекдотический пример — механик дал команду системе, управляющей полетом самолета, поднять шасси. Управляющая система выполнила команду механика несмотря на то, что самолет был на земле!

При разработке систем, критических по обеспечению безопасности, используется специальная терминология, приведенная в табл. 16.2. Эти термины взяты из работы [214] и уточнены мною.

Таблица 16.2. Терминология безопасности

Термин	Описание
Авария (или несчастный случай)	Незапланированное событие или последовательность событий, приводящие к человеческой смерти или ранению; нанесение ущерба собственности или окружающей среде. Пример несчастного случая — нанесение увечий оператору машиной, управляемой компьютерной системой
Опасность (опасные ситуации)	Ситуации, при которых возможны несчастные случаи и аварии. Пример опасности — отказ сенсора, который определяет наличие препятствия впереди машины
Повреждения	Оцениваются как ущерб от опасных случаев. Повреждения могут быть как незначительными, так и катастрофическими, приводящими к гибели людей
Серьезность опасности	Оценивается по самым большим повреждениям в результате самых опасных случаев. Серьезность опасности может ранжироваться от катастрофической, приводящей к гибели людей, до незначительной
Вероятность опасности	Вероятность появления событий, которые создают опасные ситуации. Значение вероятности определяется обычным образом. Опасные события ранжируются от вероятного (если, например, вероятность равна 1/100, т.е. на 100 "нормальных" событий произойдет одно опасное) и до невозможного (когда ни при каких условиях не возникнет опасной ситуации)

Окончание табл. 16.2

Термин	Описание
Риск	Измеряется как вероятность того, что система будет причиной несчастного случая. Для оценки риска определяются вероятность опасности, серьезность опасности и вероятность того, что опасная ситуация приведет к аварии

Считается, что система безопасна, если ее эксплуатация исключает аварии (несчастные случаи) или их последствия незначительны. Этого можно достичь тремя дополняющими друг друга способами.

1. *Предотвращение опасности.* Система разрабатывается таким образом, чтобы избежать опасных ситуаций. Например, чтобы во время эксплуатации машины избежать попадания рук оператора под лезвие, в системе раскроя предусматривается обязательное одновременное нажатие двух отдельных кнопок управления.
2. *Обнаружение и устранение опасности.* Система разрабатывается таким образом, чтобы возможные опасные ситуации были обнаружены и устранены до того, как они приведут к аварии. Например, система, управляющая химическим предприятием, для предотвращения взрыва от высокого давления должна вовремя обнаружить избыточное давление и открыть предохранительный клапан, чтобы уменьшить это давление.
3. *Ограничение последствий.* Система может включать способы защиты, минимизирующие повреждения, возникающие в результате происшедшей аварии. Например, в систему управления двигателями самолета обычно включается автоматическая система огнетушения. В случае возгорания такая система позволяет предотвратить пожар и не ставит под угрозу жизнь пассажиров и экипажа.

Аварии и несчастные случаи обычно являются результатом нескольких событий, которые происходят одновременно с непредвиденными последствиями. Анализируя серьезные аварии, в работе [275] показано, что почти все они произошли из-за комбинации системных сбоев, а не вследствие отдельных сбоев. Непредвиденная комбинация сбоев приводила к отказу системы. В той же работе утверждается, что невозможно предупредить все комбинации сбоев системы и эти аварии — неизбежное следствие использования сложных систем. Программное обеспечение имеет тенденцию разрастаться и усложняться, а сложность программно-управляемых систем увеличивает вероятность аварий и несчастных случаев.

Это, конечно, не означает, что программное управление обязательно увеличивает риск, связанный с системой. Программное управление и текущий контроль могут повысить безопасность систем. Кроме того, программно-управляемые системы могут контролировать более широкий диапазон условий по сравнению, например, с электромеханическими системами. Они также довольно легко настраиваются. Они предполагают использование компьютерных средств, которым свойственна высокая надежность и которые относительно компактны. Программно-управляемые сложные системы могут блокировать опасность. Они могут поддерживать управление во вредных условиях, уменьшая количество необходимого обслуживающего персонала.

16.4. Защищенность

Это способность системы защищать себя от внешних случайных или преднамеренных воздействий. Примером внешних воздействий на систему могли бы быть компьютерные вирусы, несанкционированное использование системы, несанкционированное изменение сис-

темы или данных и т.д. Защищенность важна для всех критических систем. Без приемлемого уровня защищенности работоспособность, безотказность и безопасность системы теряют смысл, поскольку причиной повреждения системы могут быть внешние воздействия.

Это связано с тем, что все методы подтверждения работоспособности, безотказности и защищенности полагаются на неизменность системы при эксплуатации и на соответствие ее параметров первоначально установленным. Если установленная система была повреждена каким-то образом (например, если программное обеспечение было изменено в результате проникновения в систему вируса), то параметры надежности и безопасности, которые первоначально были заложены, не могут больше поддерживаться. В этом случае программное обеспечение может вести себя непредсказуемо.

Имеются определенные типы критических систем, для которых защищенность — наиболее важный показатель надежности системы. Военные системы, системы для электронной торговли и системы, включающие создание и обмен конфиденциальной информацией, должны разрабатываться с очень высоким уровнем защищенности. Например, если система резервирования билетов авиакомпании недоступна, это причиняет неудобство в связи с некоторой задержкой продажи билетов, но если система не защищена и может принимать поддельные заказы, то авиакомпания может понести большие убытки.

Существует три типа повреждений системы, которые могут быть вызваны внешними воздействиями.

1. *Отказ в предоставлении системных сервисов.* Система может быть переведена в такое состояние, когда нормальный доступ к системным сервисам становится невозможным. Очевидно, это отражается на работоспособности системы.
2. *Разрушение программ и данных.* Компоненты программного обеспечения системы могут быть несанкционированно изменены. Это может повлиять на поведение системы, а следовательно, на надежность и безопасность. Если повреждение серьезно, система может стать не пригодной к эксплуатации.
3. *Раскрытие конфиденциальной информации.* Информация, находящаяся под управлением системы, может быть конфиденциальной, внешнее проникновение в систему может сделать ее публично доступной. В зависимости от типа данных, это может повлиять на безопасность системы и вызвать дальнейшие изменения в системе, которые скажутся на ее работоспособности и безотказности.

Как и в случае с другими составляющими надежности, есть специальная терминология, связанная с защищенностью систем. Некоторые термины, определенные в монографии [279], приведены в табл. 16.3.

Таблица 16.3. Терминология защищенности

Термин	Описание
Внешнее воздействие	Воздействие, в результате которого возможна потеря данных и/или повреждение системы
Уязвимость	Дефект системы, который может стать причиной потери данных или ее повреждения
Атака	Использование уязвимости системы
Угрозы	Обстоятельства, которые могут привести к потере данных или повреждению системы
Контроль	Защитные меры, уменьшающие уязвимость системы

В терминологии защищенности есть много общего с терминологией безопасности. Так, внешнее воздействие аналогично аварии (несчастному случаю), а уязвимость – опасности. Следовательно, существуют аналогичные подходы, увеличивающие защищенность системы.

1. *Предотвращение уязвимости.* Система разрабатывается таким образом, чтобы ее уязвимость была как можно ниже. Например, система не соединяется с внешней сетью, чтобы избежать воздействия из нее.
2. *Обнаружение и устранение атак.* Система разрабатывается таким образом, чтобы обнаружить предпринятую на нее атаку и устранить ее, пока она не привела к повреждениям и потерям. Пример обнаружения атак и их устранения – использование антивирусных программ, которые анализируют поступающую информацию на наличие вирусов и устраняют их в случае проникновения в систему.
3. *Ограничение последствий.* Система разрабатывается таким образом, чтобы свести к минимуму последствия внешнего воздействия. Например, регулярная проверка системы и возможность переустановить ее в случае повреждения.

Защищеиность становится еще более актуальиой при подключении системы к Internet. И хотя Intemet-связь обеспечивает дополнительные функциональные возможности системы (например, клиент может получить удаленный доступ к своему банковскому счету), такая система может быть разрушена злоумышленниками. При подключенности к Internet уязвимые места системы становятся доступными для большого количества людей, которые могут воздействовать на систему.

Очень важным атрибутом систем, подключенных к Internet, является их жизнеспособность [107], т.е. способность системы продолжать работать, в то время как она подвергается внешним воздействиям и часть ее повреждена. Жизнеспособность, конечно, связана и с защищенностью, и с работоспособностью. Для обеспечения жизнеспособности следует определить основные ключевые компоненты системы, которые необходимы для ее функционирования [108]. Для повышения жизнеспособности используется три стратегии: противодействие внешним воздействиям, распознавание их и восстановление системы после атаки. Здесь нет возможности охватить эту тему, но на Web-странице данной книги имеются ссылки на источники с информацией относительно исследования жизнеспособности систем.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Функциональная надежность компьютерной системы – свойство, отражающее степень доверия к ней со стороны пользователей. Наиболее важные составляющие функциональной надежности: работоспособность, безотказность, безопасность и защищенность.
- Критическая система – система, в которой отказы могут приводить к значительным экономическим потерям, физическому повреждению или угрозам человеческой жизни. Три важных класса критических систем – системы, критические по обеспечению безопасности, системы, критические для целевого назначения, и системы, критические для бизнеса.
- Работоспособность системы – способность предоставлять сервисы, когда это необходимо. Безотказность системы – способность системных сервисов работать так, как предусмотрено их назначением.
- Безотказность и работоспособность рассматриваются как наиболее важные свойства функциональной надежности. Если система ненадежна, трудно гарантировать ее безопасность и защищенность, поскольку они могут быть подорваны отказами системы.

- Безотказность связана с вероятностью сбоев, возникающих при эксплуатации системы. Программа может содержать определенные ошибки, но будет восприниматься пользователями как безопасная, поскольку они могут никогда не использовать средства системы, содержащие эти ошибки.
- Безопасность — свойство системы, которое отражает ее способность функционировать без угроз людям или окружающей среде. Если безопасность является основной характеристикой, значит это система, критическая по обеспечению безопасности.
- Защищенность важна для всех критических систем. Без приемлемого уровня защищенности работоспособность, безотказность и безопасность системы не имеют большого значения, поскольку внешние воздействия могут причинить повреждения системе.

Упражнения

16.1. Перечислите наиболее важные составляющие надежности систем. Почему зависимость между стоимостью разработки системы и ее надежностью имеет экспоненциальный вид?

16.2. Почему функциональная надежность важна для критических систем? Назовите шесть причин.

16.3. Объясните на примерах трудности точного определения безотказности программных систем.

16.4. Оцените надежность какой-либо системы, которую вы регулярно используете, перечислив отказы системы и наблюдаемые сбои. Составьте руководство пользователя, в котором опишите, что нужно делать для эффективного использования системы при наличии этих сбоев.

16.5. Назовите шесть промышленных изделий, которые содержат или могут содержать в будущем программные системы, критические по обеспечению безопасности.

16.6. Объясните, почему обеспечение безотказности системы не гарантирует ее безопасности.

16.7. В медицинской системе, управляющей облучением опухолей, предусмотрите опасность, которая может возникнуть в процессе работы системы, и предложите программное средство, которое определяет эту опасность и предотвращает возможные несчастные случае, обусловленные ею.

16.8. Объясните зависимость между работоспособностью системы и ее защищенностью.

16.9. В терминах компьютерной безопасности объясните разницу между атакой на систему и угрозой ей.

16.10. Этично ли поставлять клиенту программную систему с известными ошибками? Разумно ли сообщать клиенту о существовании этих ошибок и следует ли после этого принимать претензии относительно надежности программного обеспечения?

16.11. Предположим, вы входили в состав группы разработчиков программного обеспечения для химического завода, которое сработало неправильно и стало причиной серьезного загрязнения окружающей среды. Ваш босс заявил в телевизионном интервью, что нет никаких ошибок в программном обеспечении и что проблемы возникли из-за неправильной эксплуатации системы. Вы согласитесь с таким объяснением? Обсудите, как вы должны отнестись к такому заявлению.

17

Спецификация критических систем

Цели

Цель настоящей главы – описать функциональные и нефункциональные требования, определяющие надежность программных систем. Прочитав эту главу, вы должны:

- познакомиться с количественными показателями безотказности и понять, как они используются для определения требований безотказности;
- понять, как требования безопасности для критических систем можно получить из анализа возможных опасностей и рисков;
- понять сходство процессов для определения системных требований безопасности и защищенности.

Содержание

В главах 5–9 обсуждались системные требования и методы их разработки. В этой главе рассматриваются некоторые специальные вопросы спецификации критических систем. Поскольку цена отказа программных систем, особенно критических, как правило, очень высока, необходимо гарантировать высокое качество разработки спецификации критических систем и точность отражения в ней реальных потребностей пользователей.

Условия надежности критических систем порождают как функциональные, так и нефункциональные требования.

1. Функциональные требования к системе подразумевают возможность обнаружения ошибок, восстановления системы и обеспечения ее защиты от сбоев.
2. Нефункциональные требования можно сформулировать как требования безотказности и работоспособности системы.

Анализируя безопасность и защищенность программного обеспечения, можно сформулировать еще один тип требований, которые трудно классифицировать как функциональные или нефункциональные. Их, возможно, лучше описать как требования "не делать". В противоположность обычным функциональным требованиям, которые определяют систему посредством требований "делать", условия "не делать" описывают систему "от противного", определяя ее поведение, которое недопустимо. Приведем примеры таких требований.

Система не должна позволять пользователю изменять файлы, которые он не создавал (защищенность системы).
Система не должна разрешать режим обратного хода, если самолет находится в полете (безопасность системы).
Система не должна одновременно выполнять более трех сигналов тревоги (безопасность системы).

Пользовательские требования к критическим системам всегда определяются с применением естественного языка и системных моделей. Как отмечалось в главе 9, при разработке критических систем будут рентабельны метод формальных спецификаций и соответствующий метод верификации систем [148, 348]. Формальные спецификации не только служат основой для верификации в процессе разработки и реализации программных систем. Они наиболее точно описывают системы ПО, что позволяет избежать неоднозначного толкования требований. Кроме того, для использования формальных спецификаций необходим детальный анализ требований, а это эффективный путь обнаружения и устранения проблем в системной спецификации.

17.1. Требования безотказности

Безотказность – это комплексное понятие, которое нужно рассматривать на общесистемном уровне, а не на уровне отдельных компонентов. Так как компоненты системы взаимосвязаны, отказ в одном компоненте может распространиться через систему на другие компоненты. В компьютерных системах при определении безотказности учитывают три составляющие.

1. *Безотказность аппаратных средств.* Определяется как вероятность выхода из строя аппаратных средств и количество времени, затрачиваемого на их ремонт или замену.
2. *Безотказность программного обеспечения.* Определяется как вероятность сбоев в работе ПО.
3. *Безотказность системного оператора.* Определяется как вероятность ошибки, допущенной системным оператором.

Все эти составляющие тесно связаны. Сбой оборудования может служить причиной появления ложного сигнала на входе программного компонента. Программное обеспечение после этого может вести себя непредсказуемо. Непредвиденное поведение системы может привести в замешательство оператора и быть причиной его напряженного состояния. В условиях стресса ошибка оператора весьма вероятна. Оператор может предпринять действия, не соответствующие возникшей ситуации. Эти неверные действия могут привести к другим ошибкам в работе системы. Таким образом, может возникнуть ситуация, когда простой сбой подсистемы, которую можно быстро восстановить, приведет к серьезным проблемам, требующим полного перезапуска системы.

В настоящее время в инженерии программного обеспечения выделяется самостоятельная дисциплина, которая занимается проблемами создания надежных и безотказных программных систем [227, 20*, 26*, 29*]. В рамках этой дисциплины подсчитываются вероятности сбоя различных системных компонентов и определяется, как их сочетания влияют на общую безотказность системы. Упрощенно, если в системе присутствуют компоненты A и B с вероятностями отказа P_A и P_B, то вероятность отказа системы P_S будет такова: $P_S = P_A + P_B$.

При возрастании числа зависимых компонентов вероятность отказа системы также возрастает. Если в системе очень много критических компонентов, то каждый компонент в отдельности должен быть очень надежным для того, чтобы вероятность P_S была низкой. Для увеличения надежности компоненты могут дублироваться (см. главу 18). Тогда группа одинаковых компонентов, дублирующих друг друга, будет работать корректно так долго, пока хотя бы один компонент будет работать правильно. Это означает, что, если вероятность отказа отдельного компонента равна P_A и все отказы независимы, вероятность отказа P_S этой группы компонентов будет $P_S = P_S^n$.

Безотказность системы можно определить как нефункциональное требование, которое численно выражается через показатели, обсуждаемые в следующем разделе. Для выполнения нефункциональных требований безотказности необходимо дополнительно задать функциональные требования к системе, определяющие способы исключения системных сбоев. Примеры таких требований следующие.

1. Установление определенного диапазона для всех величин, вводимых оператором, и системный контроль всех вводимых величин для проверки, попали ли они в этот диапазон.
2. Во время процесса инициализации система должна проверить все диски на наличие сбойных блоков.
3. Для реализации подсистемы управления остановом системы следует привлекать N-вариантное программирование (специальный метод обеспечения отказоустойчивости ПО).
4. Система должна быть реализована в безопасном подмножестве языка Ada и проверена с использованием статического анализа (см. главу 19).

Не существует простых правил, которые можно использовать для получения функциональных требований безотказности. Организации — разработчики критических систем обычно имеют определенные знания о возможных требованиях безотказности и о том, как эти требования влияют на фактическую безотказность системы.

17.1.1. Показатели безотказности

Первоначально показатели безотказности были разработаны для аппаратных компонентов. Отказ отдельных аппаратных компонентов неизбежен из-за физических факторов: механический износ, электрический нагрев и т.д. Компоненты имеют определенный срок службы, поэтому наиболее широко используемым показателем безотказности оборудования является среднее время его безотказной работы. При отказе аппаратного компонента (особенно, если отказы часты) важным показателем является среднее время восстановления, показывающее время его ремонта или замены.

Из-за различной природы сбоев программного обеспечения и оборудования показатели надежности аппаратных средств не всегда приемлемы для описания требований безотказности программного обеспечения. Сбои в работе программных компонентов — это явления скорее случайные, чем постоянные. Обычно они проявляются только при определенных входных воздействиях. Если данные не повреждены, система чаще всего может продолжать работать, даже когда произошел сбой.

В табл. 17.1 приведены показатели, которые используются для определения безотказности и работоспособности программного обеспечения. Выбор показателей зависит от типа системы ПО и области ее применения.

Таблица 17.1. Показатели безотказности

Показатель	Объяснение
Вероятность отказа	Вероятность отказа в работе системы. Значение вероятности отказа 0,001 означает, что сбой произойдет один раз на тысячу случаев нормальной работы системы
Частота отказа	Значение частоты отказа 2/100 означает, что на каждые 100 единиц времени работы системы могут произойти два отказа. Этот показатель иногда называют интенсивностью отказов
Среднее время безотказной работы	Это среднее время между двумя последовательными сбоями. Значение 500 этого показателя означает, что сбой может ожидаться каждые 500 единиц времени
Работоспособность	Вероятность готовности системы к использованию. Значение работоспособности 0,998 означает, что на каждые 1000 единиц времени система будет готова к работе в 998 случаях

Приведем для каждого показателя безотказности типы систем, к которым они могут применяться.

1. *Вероятность отказа.* Наиболее подходит для систем, время функционирования которых или заранее не определено, или велико, причем отказ в системе может иметь серьезные последствия. Примерами могут служить специальные защитные системы, в частности контроля на химическом производстве или аварийной остановки в энергосистемах.
2. *Частота отказа.* Подходит для систем, от которых требуется регулярная длительная безотказная работа. Этот показатель можно использовать в требованиях, предъявляемых к банковской системе, обрабатывающей счета клиентов, или к системе бронирования мест в гостинице.

3. *Среднее время безотказной работы.* Может использоваться в системах, которые обрабатывают большие объемы данных, при этом время между отказами должно быть больше среднего времени обработки транзакций. Примеры систем, где этот показатель может использоваться: текстовый редактор и автоматизированные системы проектирования.
4. *Работоспособность.* Должен использоваться в системах, предназначенных для непрерывной работы. Примеры таких систем: телефонные коммутаторы и системы сигнализации на железной дороге.

Существует три вида числовых показателей, которые можно использовать при оценке безотказности системы.

1. Число сбоев системы для заданного периода работы. Используется для вычисления вероятности отказа.
2. Время (или количество транзакций) между сбоями системы. Используется для вычисления частоты отказа и среднего времени безотказной работы.
3. Время ремонта или время на восстановление работоспособности системы после сбоя. Используется для измерения работоспособности.

Единицы измерения, которые могут использоваться при измерении этих числовых показателей, — календарное время, время работы процессора или, может быть, некоторая дискретная единица типа числа транзакций. В системах, которые тратят много времени на ожидание ответа на запрос, например телефонные коммутаторные системы, в качестве единицы времени должно быть использовано время работы процессора. Основной единицей измерения безотказности систем является календарное время.

Календарное время является наиболее подходящей единицей измерения времени в системах, действующих непрерывно. Примером могут служить системы аварийной сигнализации, системы текущего контроля и другие типы управляющих систем. Системы, которые оперируют транзакциями, например банкоматы и системы резервирования авиабилетов, имеют различные нагрузки при функционировании в течение дня. В этих случаях единицей измерения является число транзакций; тогда частота отказа будет равна числу сбойных транзакций, отнесенных к общему количеству обработанных транзакций.

17.1.2. Нефункциональные требования безотказности

Во многих системных спецификациях требования безотказности тщательно не выписаны. Часто требования безотказности субъективны и неизмеряемы. Например, утверждение "программное обеспечение должно быть безотказным при нормальных условиях эксплуатации" бессмысленно. Квазиколичественные утверждения "программное обеспечение должно иметь не более N сбоев на 1000 строк программного кода" бесполезны. Невозможно измерить число сбоев на 1000 строк кода, так как нельзя сказать, когда эти сбои будут обнаружены. Эти утверждения ничего не говорят о поведении системы в процессе работы.

Типы отказов, происходящие в системах, а также их последствия, зависят от природы этих отказов. При разработке требований безотказности необходимо идентифицировать различные типы отказов и определить, как они должны обрабатываться. Типы системных отказов приведены в табл. 17.2.

Таблица 17.2. Классификация отказов

Отказы	Описание
Случайные	Происходят только при определенных входных данных или сигналах
Перманентные	Происходят при всех входных данных (сигналах)
Самовосстанавливаемые	Система после таких отказов может восстановиться без вмешательства оператора
Несамовосстанавливаемые	Необходимо вмешательство оператора для устранения такого отказа системы
Не разрушающие	Отказ не разрушает систему или данные
Разрушающие	Отказ разрушает систему или данные

Многие большие системы состоят из нескольких подсистем с различными требованиями к безотказности. Так как безотказное ПО дорого, обычно благоразумнее оценить требования безотказности каждой подсистемы отдельно, чем накладывать требование безотказности на все подсистемы. Это позволяет избежать излишне высоких требований безотказности тех подсистем, где в этом нет необходимости.

Приведем последовательность шагов по определению требований безотказности.

1. Для каждой подсистемы необходимо определить возможные системные отказы и проанализировать их последствия.
2. Затем необходимо отнести отказы к соответствующему классу. В качестве отправной точки можно использовать типы отказов, приведенные в табл. 17.2.
3. Для каждого класса отказов необходимо определить требования к безотказности, используя соответствующие показатели, причем для различных классов можно использовать разные показатели.
4. Формулируются функциональные требования безотказности таким образом, чтобы уменьшить вероятность критических отказов.

В качестве примера специфицирования требований безотказности рассмотрим требования к безотказности банкоматов, объединенных в сеть. Примем во внимание, что каждый банкомат используется приблизительно 300 раз в день. Срок службы оборудования — восемь лет, программное обеспечение обычно модифицируется каждые два года. Следовательно, до выпуска новой версии программного обеспечения, каждый банкомат обработает приблизительно 200 тыс. транзакций. Банк имеет в сети 1000 банкоматов. Это означает, что центральная база данных обрабатывает 300 тыс. транзакций в день (или 100 млн. ежегодно).

Отказы системы банкоматов подразделяются на два больших класса: те, которые относятся к отдельному банкомату, и те, которые относятся к базе данных и, следовательно, воздействуют на всю сеть банкоматов. Ясно, что последний тип отказов более опасен, чем отказы, относящиеся к отдельному банкомату.

В табл. 17.3 показаны возможные типы отказов и соответствующие показатели для этих отказов. Требования безотказности определены таким образом, что перманентный отказ возможен приблизительно один раз в три года. Это означает, что в сети банкоматов каждый день может выйти из строя в среднем один банкомат. Таким образом, сбои, в результате которых обработка транзакции прерывается и пользователь должен начать операцию сначала, случаются довольно часто. Но это относительно небольшое неудобство для пользователя.

Таблица 17.3. Требования безотказности для сети банкоматов

Тип отказа	Пример	Показатель
Постоянный неразрушающий	Система перестает функционировать с любыми данными на входе. Для устранения отказа систему необходимо перезагрузить	Частота отказа: 1 отказ за 1000 дней
Случайный неразрушающий	Данные на магнитной полосе не читаются, хотя карта не повреждена	Частота отказа: 1 отказ на 1000 карт
Случайный разрушающий	Последовательность транзакций вызывает нарушение в работе базы данных	Измерить невозможно. Во время работы системы случаться не должен

В идеале сбои, разрушающие базу данных, никогда не должны возникать за весь срок службы программного обеспечения. Поэтому требование безотказности в этом случае такое: вероятность разрушающего отказа должна быть меньше 1 отказа на 200 млн. транзакций. Тогда за весь срок службы одной версии ПО не должно произойти отказов, приводящих к разрушению базы данных. Но подобное требование безотказности фактически невозможно проверить. Пусть каждая транзакция занимает 1 секунду машинного времени и для сети банкоматов построен имитатор. Время моделирования транзакций, проходящих через сеть банкоматов за один день, равно 300 тыс. секунд. Это приблизительно 3,5 дня. Ясно, что это время можно сократить, уменьшая период обработки транзакций и используя несколько имитаторов, но в этом случае очень трудно проверить, насколько имитируемая система удовлетворяет требованиям безотказности.

Также невозможно проверить качественные требования, определяющие очень высокий уровень безотказности. Например, если система является критической по обеспечению безопасности, то за весь срок ее службы не должно произойти ни одного сбоя. Предположим, что должны быть установлены 1000 копий системы и что система "выполняется" 1000 раз в секунду. Срок службы системы составляет 10 лет. Следовательно, общее прогнозируемое число "выполнений" системы приблизительно равно $3 * 10^{14}$. Нет смысла определять частоту отказа как один сбой на 10^{15} "выполнений" (что обеспечит безопасность системы), поскольку для обоснования такого уровня безотказности невозможно протестировать систему.

Еще одним примером требований безотказности может служить система инъекций инсулина, которая была рассмотрена в главе 16. Она делает инъекции инсулина несколько раз в день, а уровень сахара в крови определяется несколько раз в час. Поскольку система используется периодически, а последствия сбоя весьма серьезны, то наиболее подходящим показателем безотказности системы будет вероятность отказа.

Для рассматриваемой системы можно определить два типа отказов.

1. Случайные отказы, которые пользователь может устранить самостоятельно (например, перезапустив или настроив систему). Допустимая вероятность отказа не должна превышать 0,002. В этом случае на 500 запросов к системе может произойти один отказ. Это означает, что возможен сбой приблизительно один раз в 3,5 дня.
2. Перманентные отказы, которые требуют восстановления системы разработчиком. Вероятность такого типа отказа должна быть намного ниже — приблизительно один раз в год. В этом случае вероятность отказа должна быть не больше чем 0.00002.

Стоимость разработки и проверки требований безотказности системы может быть очень высока. Организации должны реально оценивать, необходимы ли такие затраты. Они оправданы в системах, где безотказность является критическим фактором, например в телефонных коммутаторах, или в системах, в которых отказ может привести к большим экономическим потерям. Вероятно, они не оправданы для многих типов систем, применяемых в бизнесе и научных исследованиях. Такие системы обычно имеют умеренные требования к безотказности, так как отказ в них просто приводит к задержке процесса и устранить его относительно недорого.

17.2. Специфицирование требований безопасности

Безопасная работа является необходимой характеристикой систем ПО, критических по обеспечению безопасности. Это означает, что при разработке требований необходимо рассмотреть потенциальные опасности и риски, которые могут возникнуть в процессе работы системы. В требованиях следует либо описать, как программная система должна вести себя, чтобы минимизировать риск, либо предусмотреть, чтобы опасность никогда не возникала.

Процесс определения требований безопасности и обеспечение их выполнения — часть общего "безопасного" жизненного цикла ПО [290], в упрощенном виде представленного на рис. 17.1. Безопасный жизненный цикл ПО предложен в международном стандарте управления безопасностью IEC 61508 [175], разработанном Международной электротехнической комиссией (International Electrotechnical Commission — IEC). Как можно видеть из рис. 17.1, этот стандарт охватывает все аспекты управления безопасностью — от начального определения области применения через планирование и разработку системы до вывода ее из эксплуатации.

На первых стадиях жизненного цикла стандарта IEC 61508 определяется область действия системы, оцениваются потенциальные опасности и риски. Далее следует специфицирование требований безопасности и определение подсистем, отвечающих за выполнение конкретных положений требований безопасности. Затем следуют этапы планирования и реализации систем. Системы, критические по обеспечению безопасности, разрабатываются с учетом внешнего окружения, что может дать дополнительные средства снижения рисков. Параллельно с разработкой системы планируются проверка соответствия системы требованиям безопасности, ее инсталляция, эксплуатация и сопровождение.

Управление безопасностью не заканчивается с поставкой системы заказчику. После поставки система должна быть инсталлирована так, как запланировано при анализе опасностей. Прежде чем система будет запущена в эксплуатацию, проводится ее проверка на безопасность. Контроль безопасности должен осуществляться в течение всего срока эксплуатации и обслуживания системы. Много проблем, связанных с безопасностью, возникает из-за плохого обслуживания, поэтому особенно важно, чтобы система была удобной для сопровождения. В заключение следует подчеркнуть, что необходимые меры безопасности должны быть также приняты при выводе системы из эксплуатации (например, удаление из системы опасных материалов).

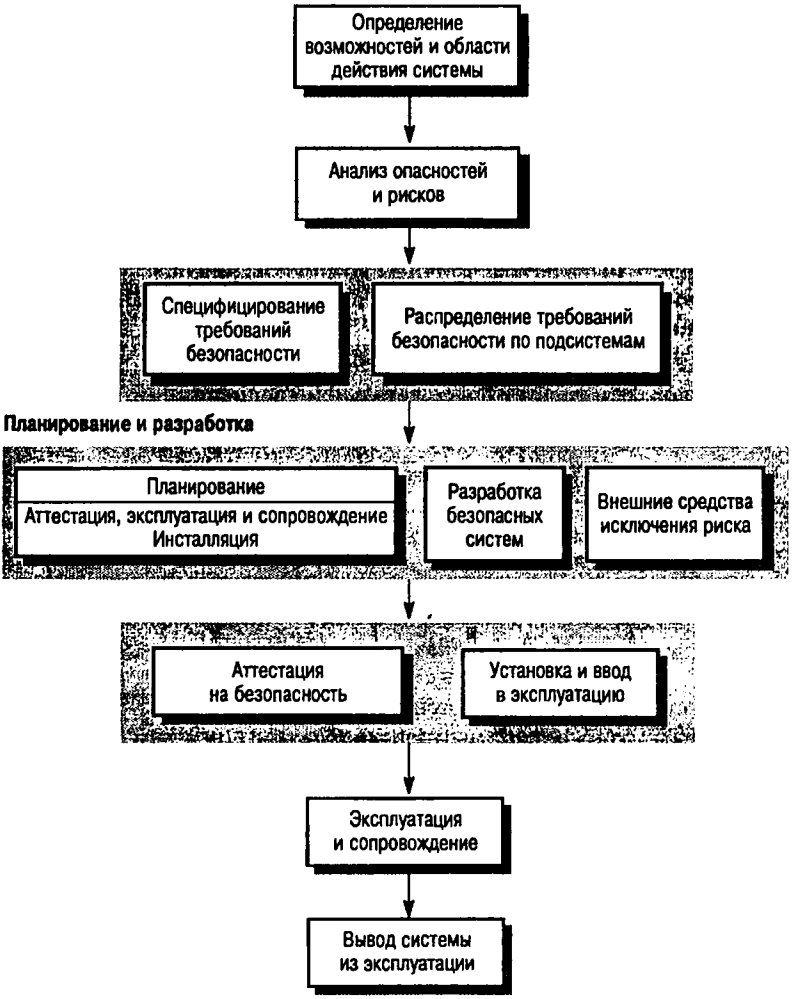


Рис. 17.1. Безопасный жизненный цикл ПО в соответствии со стандартом IEC 61508

17.2.1. Анализ опасностей и рисков

Анализ опасностей и рисков складывается из анализа самой системы и окружения, в котором она будет эксплуатироваться. В результате должны быть обнаружены потенциальные опасности, которые могут возникнуть в системе и ее окружении. Это комплексный и трудоемкий процесс, который требует всестороннего анализа и большого кругозора разработчиков. Эту работу должны проводить опытные инженеры совместно со специалистами в той предметной области, где будет эксплуатироваться система, и профессионалами в области безопасности систем.

На рис. 17.2 показан итерационный процесс анализа рисков и опасностей, который включает ряд этапов.

1. *Определение опасностей.* Это этап определения потенциальных опасностей, которые могут возникнуть. Они зависят от окружения, в котором система будет эксплуатироваться.
2. *Анализ рисков и классификация опасностей.* На этом этапе выявленные опасности рассматриваются каждая в отдельности. Для дальнейшего анализа выбираются те из них, которые являются потенциально серьезными и наиболее вероятными. Некоторые выявленные опасности исключаются из дальнейшего рассмотрения, поскольку они маловероятны (например, поражение от молнии при землетрясении).
3. *Анализ опасностей.* Выявляются причины возникновения опасностей. На этом этапе могут использоваться различные методы анализа, иапример анализ дерева отказов (см. следующий раздел).
4. *Оценка возможностей уменьшения рисков.* Предлагаются способы уменьшения или устранения рисков. Это требует более детальной спецификации требований безопасности, как показано в модели безопасного жизненного цикла (см. рис. 17.1).

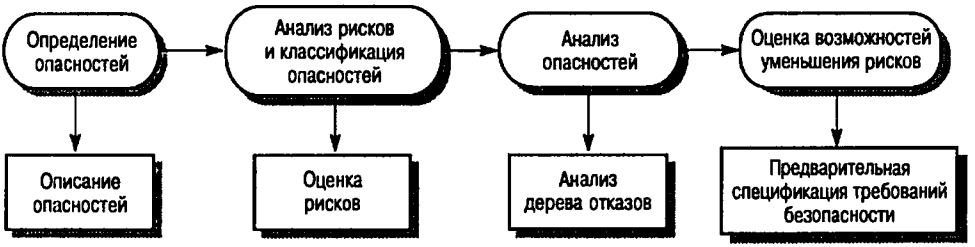


Рис.17.2. Анализ опасностей и рисков

Для больших систем анализ опасностей и рисков обычно подразделяется на несколько этапов [215]. Они включают:

- предварительный анализ опасностей, для которых определены основные риски;
- более детальный анализ системы и подсистем для выявления скрытых опасностей;
- анализ компонентов ПО, для которых рассматриваются риски отказов;
- анализ опасностей на этапе эксплуатации, имеющих отношение к пользовательскому интерфейсу системы, и риска, возникающего в результате ошибок оператора.

Этот анализ определяет потенциальные опасности и с каждой из них связывается риск. Здесь оцениваются вероятности возникновения опасностей, причинения ущерба и вероятности серьезного ущерба.

Процесс анализа опасности обычно включает рассмотрение различных классов опасностей, например физических, электрических и биологических, опасностей облучения, опасностей из-за плохого обслуживания системы и т.д. Каждый из этих классов затем тщательно анализируется, чтобы обнаружить взаимосвязанные опасности.

Для уже упоминавшейся системы инъекций инсулина опасности и классы, которым они принадлежат, перечислены ниже.

1. Передозировка инсулина (ошибка обслуживания).
2. Недостаточная доза инсулина (ошибка обслуживания).
3. Сбой питания из-за разрядки аккумулятора (электрическая опасность).

4. Влияние электрического поля другого медицинского оборудования, например кардиостимулятора, на систему (электрическая).
5. Неправильная установка датчика или плохой контакт (физическая).
6. Частичное извлечение иглы из тела пациента (физическая).
7. Инфекция, вызванная введением иглы (биологическая).
8. Аллергическая реакция пациента на материалы или инсулин (биологическая).

Риски, связанные с этими опасностями, описаны в разделе 17.2.3. Так как данная система небольшая, нет необходимости в многоэтапном анализе опасностей. Но многие системы, критические по обеспечению безопасности, очень большие (например, химический завод), и анализ опасностей для таких систем – длительный, сложный и дорогостоящий процесс.

17.2.2. Анализ дерева отказов

Для каждой выявленной опасности необходимо выполнить детальный анализ, чтобы определить условия, при которых возникает эта опасность. Различают дедуктивные и индуктивные методы анализа опасностей. Дедуктивные методы более просты в использовании и заключаются в анализе работы системы, которая запускается в условиях возникновения опасности и работает до момента отказа. В индуктивных методах запускается система в условиях предполагаемого отказа и определяются возникающие опасности. По возможности для анализа опасностей должны использоваться и индуктивные и дедуктивные методы.

В рамках индуктивных и дедуктивных методов существуют различные подходы к анализу опасностей и рисков. Они включают экспертизы таблиц контрольных проверок и такие более формальные методы, как анализ сетей Петри [277], методы формальной логики [188] и анализ дерева отказов [183].

Здесь я описываю анализ дерева отказов. Это широко используемая и сравнительно простая методика анализа опасностей, ее можно применять без знаний предметной области. Анализ дерева отказов начинается с описания опасности и далее осуществляется продвижение по дереву отказов для определения возможных причин опасности. Таким образом, описание опасности находится в корне дерева, а причины опасности соответствуют листьям дерева.

На рис. 17.3 показано дерево отказов для возможных опасностей, связанных с ПО, в системе инъекций инсулина. Здесь опасность "неправильное управление дозой инсулина" представляет серьезную угрозу для пациента. Конечно, программное обеспечение должно отличать опасность передозировки инсулина от опасности недостаточной дозы и по-разному реагировать на них.

Дерево отказов на рис. 17.3 не завершено. Здесь показаны только потенциальные сбои и ошибки программного обеспечения. Сбои оборудования, например понижение напряжения батареи, вызывающие отказ датчика, не показаны. На этом уровне детализации дальнейший анализ опасностей невозможен. Чтобы использовать дерево отказов для анализа опасностей, возникающих в процессе проектирования и разработки систем, необходима более детальная и подробная информация. В работах [183, 214] показано, как деревья отказов можно использовать для анализа программного обеспечения вплоть до уровня отдельных операторов программного кода.

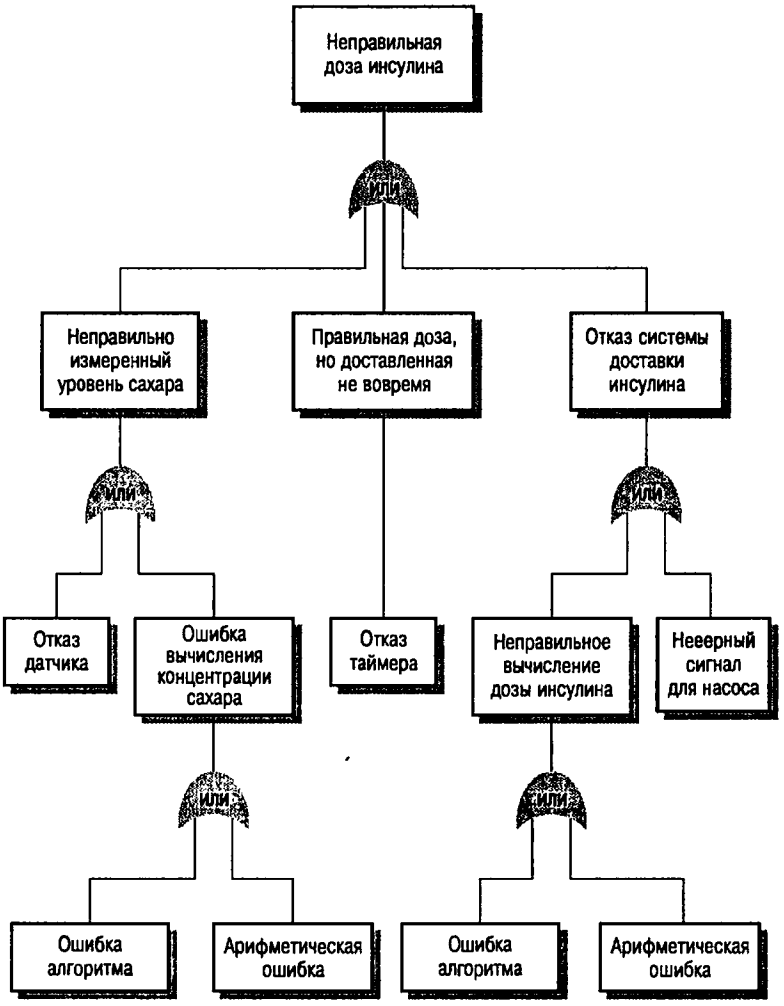


Рис.17.3. Дерево отказов системы инъекций инсулина

17.2.3. Оценка рисков

Процесс оценивания рисков начинается после того, как все опасности определены и описаны. При оценке риска рассматривается серьезность каждой опасности, вероятность ее возникновения и тех последствий, которые могут быть ею вызваны. В результате анализа риска для каждой опасности оценивается ее допустимость (приемлемость). Обычно опасности классифицируются (по критерию допустимости) следующим образом.

1. *Недопустимые.* В этом случае система разрабатывается таким образом, чтобы опасность либо не возникала, либо, если возникла, не приводила к тяжелым последствиям.

2. *Минимально допустимые.* В этом случае система разрабатывается так, чтобы вероятность тяжелых последствий была минимальна, при этом должна учитываться стоимость и сроки такой разработки.
3. *Допустимые.* Разработчики системы, уменьшая вероятность возникновения таких опасностей, не должны увеличивать стоимость и сроки разработки системы.

На рис. 17.4 показаны три области, соответствующие перечисленным видам опасностей [57]. Форма диаграммы отражает затраты на обеспечение безопасности, здесь стоимость проектирования безопасных систем пропорциональна длине основания треугольника. Самые высокие затраты для устранения опасности – в верхней части диаграммы, минимальные затраты на обеспечение безопасности – у вершины треугольника.

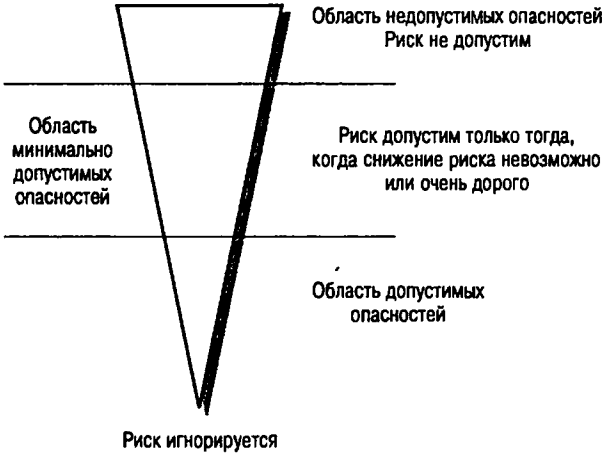


Рис. 17.4. Уровни риска

Границы между областями на рис. 17.4 не фиксированы и имеют тенденцию со временем перемещаться вниз в силу изменения общественного мнения и политических соображений. Хотя финансовые затраты на устранение последствий опасностей могут быть меньше стоимости превентивных мер безопасности, общественное мнение может потребовать, чтобы дополнительные затраты на обеспечение безопасности были приняты. Например, часто для компаний дешевле устранить загрязнение, которое может возникнуть с малой долей вероятности, чем устанавливать систему для предотвращения загрязнения. Это было допустимо в 1960–1970-х годах, но в настоящее время общественное и политическое мнение такого допустить не может. Граница между областями недопустимых и минимально допустимых опасностей передвинулась вниз так, что опасности, которые в прошлом были допустимы, сейчас стали недопустимыми.

Процесс оценки риска заключается в оценке вероятности опасности и серьезности опасности. Обычно трудно сделать это точно, поэтому часто используют относительные оценки типа "вероятно", "маловероятно", "редко", а также "высокая", "средняя" и "низкая". Иногда с этими оценками можно связать числовые значения. Но поскольку серьезные последствия опасностей случаются сравнительно редко, практически невозможно проверить точность этих значений.

В табл. 17.4 показаны риски для опасностей, определенных в предыдущем разделе для системы инъекций инсулина. Так как я не врач, оценки риска в этой таблице служат толь-

ко для иллюстрации. Например, известно, что передозировка инсулина потенциально более опасна, чем недостаточная доза.

Таблица 17.4. Анализ рисков

Опасность	Вероятность опасности	Серьезность опасности	Риск	Допустимость опасности
Передозировка инсулина	Средняя	Высокая	Высокий	Недопустима
Недостаточная доза инсулина	Средняя	Низкая	Низкий	Допустима
Отказ элетропитания	Высокая	Низкая	Низкий	Допустима
Неправильно собран механизм введения инсулина	Высокая	Высокая	Высокий	Недопустима
Повреждение механизма введения инсулина	Низкая	Высокая	Средний	Минимально допустима
Инфекция через механизм введения	Средняя	Средняя	Средний	Минимально допустима
Электрические помехи	Низкая	Высокая	Средний	Минимально допустима
Аллергическая реакция	Низкая	Низкая	Низкий	Допустима

17.2.4. Уменьшение рисков

Если потенциальные опасности и их причины определены, системные требования формулируются таким образом, чтобы опасности, приводящие к тяжелым последствиям, были маловероятны. В главе 16 я определил три стратегии, которые позволяют это сделать.

1. *Предотвращение опасности.* Система проектируется так, чтобы опасность не могла возникнуть.
2. *Обнаружение и устранение опасности.* Система разрабатывается таким образом, чтобы опасности обнаруживались и нейтрализовались раньше, чем они приведут к серьезным последствиям.
3. *Ограничение последствий.* Система разрабатывается так, чтобы свести последствия опасностей к минимуму.

Обычно разработчики систем, критических по обеспечению безопасности, используют комбинацию этих подходов. Например, в случае обнаружения недопустимых опасностей можно уменьшить вероятности их возникновения и добавить защитные системы, срабатывающие при их возникновении.

Рассмотрим ошибки программного обеспечения (см. рис. 17.3), приводящие к опасностям в системе инъекций инсулина.

1. *Арифметическая ошибка.* Она определяется как некоторое арифметическое вычисление, которое является причиной отказа системы. В спецификации требований должны быть определены все возможные арифметические ошибки, приводящие к

отказу системы. Они, конечно, зависят от используемых алгоритмов вычислений. В спецификации должен быть указан обработчик исключительных ситуаций для всех идентифицированных арифметических ошибок. В спецификации также описываются действия, которые необходимо предпринять в случае возникновения каждой из этих ошибок. При возникновении ошибок система должна остановиться и включить аварийную сигнализацию.

2. *Алгоритмическая ошибка.* Это более сложная ошибка, так как аномальную ситуацию трудно обнаружить. Ее можно было бы обнаружить, сравнивая требующуюся дозу инсулина с дозой, рассчитанной заранее. Если доза велика, то это значит, что она вычислена неправильно. Система может также следить за последовательностью доз. Если число введенных доз выше среднего, то может быть выдано предупреждение и ограничена дальнейшая дозировка.

Для определения возможных проблем оборудования используется дерево отказов. Это помогает понять требования к программному обеспечению для выявления и, возможно, исправления потенциальных проблем. Например, инсулиновые дозы не назначаются с большой частотой: не больше двух или трех раз в час, а иногда и меньше этого. Поэтому необходимы диагностические и управляющие программы. Ошибки оборудования (датчика, насоса или таймера) должны быть обнаружены, а предупреждения выданы еще до того, как они серьезно повлияют на пациента.

Некоторые требования безопасности для системы инъекций инсулина приведены во врезке 17.1. Они являются пользовательскими требованиями и, конечно, должны быть детализированы в заключительной системной спецификации. Табл. 3 и 4, на которые есть ссылки в требованиях, должны быть включены в документ спецификации.

Врезка 17.1. Примеры требований безопасности для системы инъекций инсулина

Требование 1	Разовая доза инсулина не должна превышать максимальной дозы, определенной для пользователя
Требование 2	Суммарная доза инсулина не должна превышать суммарной дозы, определенной для пользователя
Требование 3	Система должна включать диагностическое оборудование не менее 4 раз в час
Требование 4	Система должна иметь обработчик исключений, определенный в табл. 3
Требование 5	При обнаружении любой неисправности оборудования должен звучать звуковой сигнал и выводиться диагностическое сообщение, определенное в табл. 4

17.3. Спецификирование требований защищенности

Спецификация требований защищенности имеет много общего с требованиями безопасности. Требования защищенности также трудно определить количественно, часто это требования "не делать", т.е. они определяют недопустимое поведение системы, а не ее функциональные возможности. Однако есть и важные различия между этими типами требований.

1. Безопасный жизненный цикл ПО, который охватывает все аспекты управления безопасностью, разработан. Аналогичный жизненный цикл для управления защищенностью программных систем пока не разработан.

2. Набор угроз защищенности в системах довольно обобщенный. Все системы должны защищать себя от вторжения, от отказов в обслуживании и т.д. В противоположность этому опасности в системах, критических по обеспечению безопасности, обычно очень конкретны.
3. Методы и технологии защищенности (такие, как шифрование и опознавательные устройства) только появляются. Однако много технологий защищенности было разработано для специализированных систем (военных и финансовых), и сегодня существуют только проблемы переноса их на системы общего использования. Методы безопасности программного обеспечения еще представляют предмет исследования.

Обычный (не автоматизированный) приближенный анализ защищенности основан на анализе активов (ценностей), которые должны быть защищены, и стоимости организации защищенности. Поэтому высокая защищенность будет обеспечена для систем в тех областях, где размещены большие денежные суммы, в противоположность тем, где суммы ограниченны. Возможный процесс специфицирования требований защищенности показан на рис. 17.5.

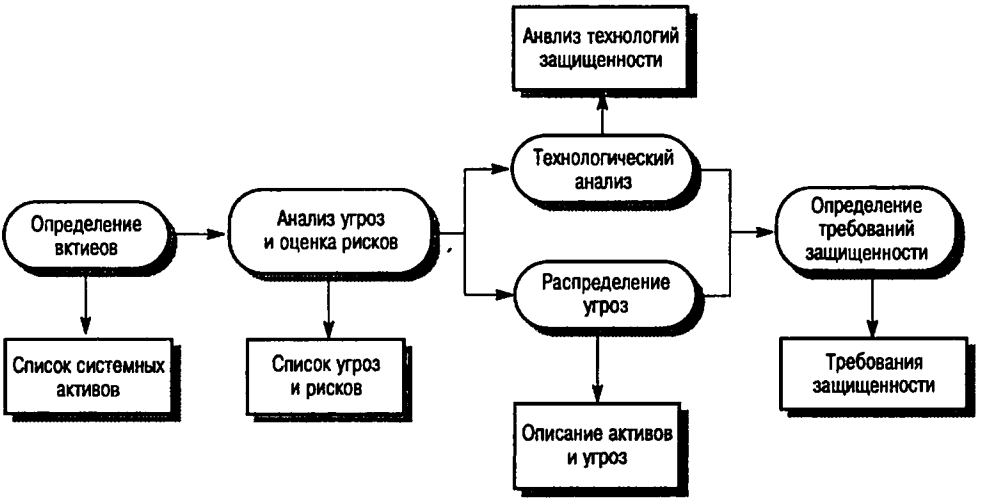


Рис. 17.5. Специфицирование требований защищенности

Этот процесс включает несколько этапов.

1. *Определение и оценка активов.* Определяются активы (данные и программы) и необходимая степень их защиты. Следует заметить, что степень требуемой защиты зависит от значимости активов, поэтому файловый пароль, например, более важен, чем множество Web-страниц, поскольку несанкционированный доступ к паролю может иметь серьезные последствия для системы.
2. *Анализ и оценка угроз и рисков.* Определяются возможные угрозы для защищенности системы и оцениваются риски, связанные с этими угрозами.
3. *Распределение угроз.* Для каждого актива определяется свой список угроз.
4. *Технологический анализ.* Оцениваются возможные технологии защиты и их применимость против идентифицированных угроз.

5. *Специфицирование требований защищенности.* Определяются требования к защищенности системы.

Определение защищенности и управление ею являются сейчас важными областями исследования в инженерии программного обеспечения. Стандарты для управления защищенностью находятся в развитии [180] и, вероятно, будут согласованы в течение нескольких ближайших лет.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Требования безотказности должны быть определены количественно в спецификации системных требований.
- Существуют различные показатели безотказности, например вероятность отказа, частота отказа, среднее время безотказной работы и работоспособность. Наиболее подходящий показатель для конкретной системы определяется в зависимости от типа системы и области ее применения. Для различных подсистем могут использоваться разные показатели.
- Нефункциональные требования безотказности могут привести к функциональным системным требованиям, которые определят системные функции, способные уменьшить число системных отказов и, следовательно, увеличить безотказность.
- Анализ опасностей является основой в процессе определения требований безопасности. Системные требования должны формулироваться таким образом, чтобы гарантировать, что эти опасности не проявятся или если они произойдут, то не приведут к тяжелым последствиям.
- Анализ рисков — это процесс оценки вероятности, что опасность приведет к тяжелым последствиям. Анализ рисков включает определение критических опасностей, которых нужно избегать, и классифицирует риски согласно их серьезности.
- Для формулирования требований защищенности необходимо определить активы, которые должны быть защищены, а также методы и технологии защищенности, которые будут использованы для защиты этих активов.

Упражнения

17.1. Почему нельзя использовать показатели безотказности аппаратных средств при оценке безотказности программных систем? Проиллюстрируйте ответ примером.

17.2. Укажите подходящие показатели безотказности для следующих программных систем. Обоснуйте ваш выбор показателя и оцените его значение.

- Система контроля за состоянием пациентов в больнице.
- Текстовый процессор.
- Автоматическая система управления торговым автоматом.
- Система торможения в автомобиле.
- Система управления холодильником.
- Генератор отчетов.

17.3. Допустим, что вы ответственны за написание спецификации программной системы, управляющей сетью кассовых терминалов в магазине. Система принимает от терминала информацию, считанную с полосы штрихового кода, обращается в базу данных товаров и возвращает на терминал для отображения наименование товара и его цену. Система должна быть постоянно работоспособна в течение рабочего времени магазина.

Выберите подходящий показатель безотказности для такой системы и напишите спецификацию требований безотказности, принимая во внимание, что возможные системные отказы и сбои имеют разную значимость.

17.4. Укажите четыре функциональных требования для системы кассовых терминалов, которые могли бы повысить безотказность системы.

17.5. Объясните, почему границы в треугольнике риска, показанном на рис. 17.4, изменяются со временем в соответствии с изменением социальных ориентиров.

17.6. В системе инъекций инсулина пользователь может управлять заменой иглы и частотой инъекций, также может менять максимальную разовую дозу и максимальную ежедневную дозу. Назовите три ошибки пользователя, которые могут произойти, и определите меры безопасности, которые позволят избежать ошибок, приводящих к несчастному случаю.

17.7. Система (критическая по обеспечению безопасности) для управления лечением пациентов, больных раком, имеет две основные составляющие.

- Излучатель, который доставляет управляемые дозы излучения к участкам опухоли. Этим устройством управляет встроенная система ПО.
- База данных лечения, которая содержит описание особенностей лечения каждого пациента. Параметры облучения также введены в эту базу данных и автоматически загружаются в излучатель.

Определите три опасности, которые могут возникнуть в этой системе. Для каждой опасности укажите меры (и обоснуйте их), которые уменьшат вероятность опасности, приводящей к несчастному случаю.

17.8. Укажите, как можно изменить анализ дерева отказов для определения требований защищенности.

18 Разработка критических систем

Цели

Цель настоящей главы – дать обзор методов разработки критических систем. Прочитав эту главу, вы должны:

- знать о методах разработки программного обеспечения, которые помогают избежать сбоев в программной системе;
- познакомиться с концепцией отказоустойчивости программного обеспечения и с тем, как безопасное программирование может повысить устойчивость систем к отказам;
- выяснить, каковы средства обработки исключительных ситуаций, используемые для реализации отказоустойчивых систем;
- знать о N-вариантном программировании и блоках восстановления – двух разных подходах к реализации отказоустойчивых архитектур.

Содержание

Внедрение в практику современных методов разработки программного обеспечения, использование новых языков программирования и применение управления качеством привели к значительному повышению надежности ПО. Однако для критических систем, таких, как системы автоматического управления, телекоммуникационные системы или системы управления двигателем самолета, необходимы дополнительные меры для достижения высокого уровня надежности. Для таких систем используются специальные методы программирования, гарантирующие безопасность, защищенность и безотказность программных систем.

Существует два дополняющих друг друга подхода, которые используются при разработке надежного программного обеспечения.

1. *Предотвращение сбоев.* В процессе проектирования и реализации используются такие технологии разработки ПО, которые сводят к минимуму ошибки оператора и помогают находить системные ошибки (прежде чем система будет запущена в эксплуатацию).
2. *Устойчивость к сбоям.* Система проектируется таким образом, чтобы можно было обнаружить и исправить сбои и непредвиденное поведение системы до того, как это приведет к отказу в ее работе.

Предотвращение сбоев означает поставку заказчику программных систем, свободных от ошибок и сбоев. Это можно сделать двумя способами: с помощью методов программирования, которые минимизируют число ошибок, возможных в системе (минимизация ошибок и сбоев); с помощью статических и динамических методов тестирования, которые обнаруживают эти ошибки и позволяют их исправить до начала эксплуатации системы (обнаружение ошибок и сбоев). В этой главе описаны методы минимизации ошибок и повышения отказоустойчивости систем, вопросы тестирования систем и обнаружения ошибок и сбоев рассматриваются в главах 19–21.

18.1. Минимизация ошибок и сбоев

Процесс создания надежного программного обеспечения преследует цель разработки *безотказного* ПО, т.е. такого, которое точно соответствует спецификации системных требований. Но точное соответствие системы своей спецификации не гарантирует, что ПО всегда будет вести себя так, как ожидается пользователями. В спецификации могут быть ошибки, которые отразятся в программном обеспечении, либо пользователи могут неверно истолковывать или неправильно эксплуатировать систему. Безотказное ПО не обязательно гарантирует отсутствие отказов в работе системы. Но, с другой стороны, минимизация ошибок программного обеспечения значительно уменьшает число отказов системы и должна выполняться при разработке критических систем.

Существует ряд требований к разработке безотказного программного обеспечения.

1. Должна быть точная (предпочтительно формальная) спецификация системных требований, определяющая разрабатываемую систему.
2. Организация – разработчик ПО должна иметь высокую культуру управления качеством, поскольку качество является главным в процессе создания критических систем. В идеале предполагается, что программисты создают программы, в которых отсутствуют ошибки.

3. Методы проектирования и реализации ПО должны основываться на сокрытии и инкапсуляции информации. Объектно-ориентированные языки, такие как Java, удовлетворяют этому условию.
4. В процессе реализации программного кода должны использоваться языки программирования со строгим контролем типов данных, например Java или Ada. В таких языках многие ошибки программирования будут обнаружены на этапе компилирования программ.
5. Везде, где возможно, следует избегать использования тех программных конструкций, которые потенциально могут привести к ошибкам. Такие конструкции обсуждаются в следующем разделе.
6. Должна быть определена четкая технология разработки ПО, и разработчики должны быть обучены применению этой технологии. Менеджеры, отвечающие за качество, должны проверять процесс разработки.

Если при разработке программ использовались языки программирования низкого уровня с ограниченным контролем типов данных, такие как C, то достигнуть безотказности программного обеспечения очень трудно. На это имеются следующие причины.

1. Эти языки включают конструкции (такие, как указатели), которые, как известно из опыта, приводят к ошибкам. Независимо от того, сколько программист затратит усилий, в программе возможны ошибки, которые очень трудно обнаружить.
2. Природа этих языков такова, что они ведут к компактному стилю программирования. Это делает программы более трудными для чтения и понимания, что усложняет поиск ошибок по тексту программ.

Конечно, преимущество использования языков низкого уровня в том, что их конструкции менее абстракты, и поэтому есть возможность написать весьма эффективные программы. В некоторых случаях высокая эффективность существенна и не может быть достигнута другим способом. Но, если требуется высокий уровень функциональной надежности ПО, придется приложить больше усилий для тестирования системы и обнаружения ошибок.

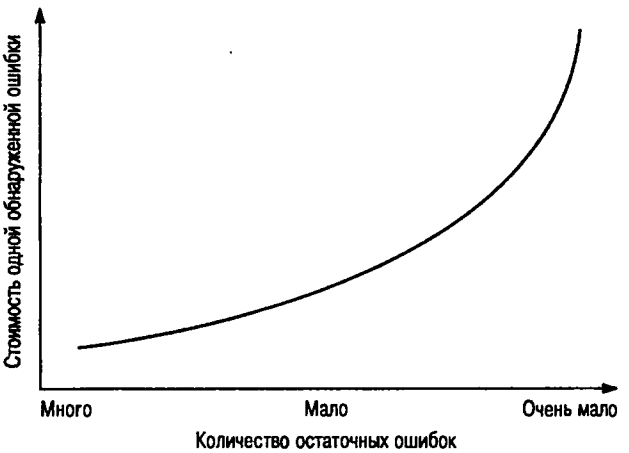


Рис. 18.1. Возрастание стоимости обнаружения ошибок

Я убежден, что современные методы разработки ПО позволяют создавать безотказное программное обеспечение, но экономически не выгодны. Чрезвычайно трудно и дорого достичь этой цели. Стоимость обнаружения и удаления программных ошибок растет экспоненциально (рис. 18.1). Поэтому организации-разработчики явно или неявно понимают, что их программное обеспечение будет содержать некоторые "остаточные" ошибки. Количество ошибок также зависит от типа систем.

Разумное объяснение наличия ошибок в системе заключается в том, что обычно дешевле устранить последствия отказа системы, чем выявить и устранить все ошибки в системе до начала ее эксплуатации. Это распространенная позиция поставщиков программных продуктов для персональных компьютеров. Однако, как указывалось в главе 16, решение о выпуске программного продукта с ошибками основывается не просто на экономических соображениях – необходимо принять во внимание социальную и политическую приемлемость последствий отказа системы.

18.1.1. Предотвращение ошибок

Ошибки в программах и, как следствие, сбои в работе систем часто являются результатом ошибок человека. Программисты делают ошибки, потому что теряют связь между взаимоотношениями переменных состояний. Они пишут программы, которые приводят к непредвиденному поведению и непредвиденному изменению состояний системы. Люди будут ошибаться всегда, но, как стало ясно в конце 1960-х, некоторые подходы к программированию более подвержены ошибкам по сравнению с остальными.

В работе [96] показано, что использование оператора безусловного перехода goto приводит к логическим конструкциям, существенно подверженным ошибкам программирования. Они также затрудняют локализацию изменений состояния системы. Это наблюдение привело к появлению так называемого *структурного программирования*. Речь идет о программировании без использования оператора goto, использующем для управление только циклы по условию while и условные операторы if. Переход к структурному программированию был важной вехой в развитии инженерии программного обеспечения, поскольку это был первый шаг на пути к научному подходу в практике разработки ПО.

Кроме операторов безусловного перехода, существуют другие языковые конструкции и методы программирования, подверженные ошибкам.

1. *Числа с плавающей запятой.* Эти числа неточны по своей природе. Это порождает определенные проблемы, особенно при сравнении чисел. Например, число 3.00000000 может иногда представляться как 2.99999999, а иногда как 3.00000001. Сравнение последних чисел показало бы, что они не равны. Числа с фиксированной запятой, где установлено количество десятичных знаков, более "надежны" при точном сравнении.
2. *Указатели.* Это низкоуровневые конструкции, содержащие адреса, которые являются ссылками непосредственно на ячейки оперативной памяти машины. Они опасны, поскольку допускают совмещение имен, а также затрудняют проверку массивов и других структур.
3. *Динамическое распределение памяти.* В этом случае выделение памяти для программы осуществляется во время ее выполнения, а не во время компиляции. Опасность кроется в возможности такого распределения памяти, при котором программа будет выполняться вне доступной области памяти. Этот тип ошибки очень трудно обнаружить, поскольку система может успешно работать в течение длительного времени, прежде чем возникнут какие-либо проблемы.

4. *Параллельность процессов.* Параллельное выполнение процессов опасно тем, что трудно предсказать взаимовлияние от обмена данными между процессами, которое может быть разным в зависимости от временных параметров обмена. Эту проблему невозможно обнаружить просмотром текста программ; кроме того, во время испытаний системы не всегда можно выявить комбинации временных параметров, которые приводят к непредвиденному взаимовлиянию процессов. Если без параллельности процессов невозможно обойтись, необходимо минимизировать взаимозависимости между процессами. В некоторых языках программирования (например, Ada и Java) имеются средства для управления параллельным выполнением процессов.
5. *Рекурсия.* Это ситуация, когда процедура или метод вызывает себя или другую процедуру, которая затем вызывает первоначальную процедуру. Использование рекурсии позволяет создавать короткие программы, однако в них трудно проследить логику. Поэтому ошибки программирования также трудно обнаружить. Использование рекурсии может привести к ошибкам в адресации памяти, так как создаются временные стеки переменных.
6. *Прерывания.* Это средство управления принудительным переходом к определенному разделу программы независимо от текущего ее выполнения. Опасности этого очевидны, поскольку прерывание может привести к непредвиденному прекращению выполнения какой-либо критически важной операции.
7. *Наследование.* Наследование в объектно-ориентированных языках программирования позволяет многократно использовать определенные фрагменты кода и выполнять декомпозицию задач, но это также приводит к тому, что программный код, относящийся к одному объекту, может находиться в разных разделах программы. Это затрудняет понимание поведения объекта, поэтому вероятны ошибки программирования.
8. *Совмещение имен.* Это случается, когда различные имена используются для обращения к одному и тому же программному объекту. При чтении текста программы легко пропустить момент изменения состояния объекта, для обращения к которому используется несколько имен.
9. *Ввод данных без проверки.* Некоторые системы организуют процесс ввода данных независимо от их содержания. Это просчет в защищенности системы, поскольку могут быть введены данные, которые не отвергаются системой, но могут привести к ее сбою.

Некоторые стандарты проектирования систем, критических по обеспечению безопасности, полностью запрещают использование перечисленных программных конструкций. Однако такая категоричность не всегда оправдана. Все эти конструкции и методы полезны, но они должны использоваться осторожно. Везде, где возможно, их потенциально опасные эффекты должны контролироваться, используя эти конструкции внутри абстрактных типов данных или объектов.

18.1.2. Сокрытие информации

Принцип защищенности, который принят военными организациями, гласит: "нет необходимости знать". Только тем, кому требуется знать особую часть информации для выполнения своих обязанностей, предоставляется доступ к ней. Информация, которая прямо не относится к их работе, скрывается.

В программировании аналогичный принцип должен применяться для управления доступом к данным. Программным компонентам должны быть доступны только те данные, которые им необходимы для выполнения своих функций. Доступ к другим данным должен быть закрыт с помощью правил видимости, которые есть практически в любом языке программирования. Если используется сокрытие информации, то скрытая информация не может быть разрушена компонентами системы, которые ее не используют.

Сокрытие информации наиболее просто реализуется в языке Java, в отличие от более ранних языков программирования типа C или Pascal. Эти языки не имеют конструкций инкапсуляции, таких как классы объектов, поэтому структуры данных не защищены. Другие части программы могут обращаться к данным непосредственно, что может вести к непредвиденным побочным эффектам. В объектно-ориентированных языках предусмотрены методы, которые позволяют выбирать и обновлять значения атрибутов, не предоставляя другим объектам непосредственного доступа к этим атрибутам. Таким образом, значение атрибута можно изменять независимо от других объектов, которые используют атрибут.

Этот же принцип в языке Java можно использовать для объявления интерфейса объекта независимо от его выполнения, что проиллюстрировано в листинге 18.1. Пользователи объектов типа Queue (Очередь) могут поместить элементы в очередь и выбрать их из очереди, а также запросить размер очереди. Однако в классе, который осуществляет этот интерфейс, фактическая реализация очереди скрыта путем объявления атрибутов и методов собственностью этого класса объектов.

Подобный пример сокрытия информации показан в листинге 18.2. В ситуациях, когда некоторая переменная может принимать ограниченное множество значений, эти значения могут быть объявлены как константы. Языки, подобные C++, поддерживают перечислимые типы данных, но в языке Java для этого необходимо использовать объявление класса. Для примера рассмотрим сигнальную систему, которая поддерживает красный, желтый и зеленый цвета. Тип Signal (Сигнал) необходимо определить так, чтобы включить объявления констант, соответствующих этим цветам. После этого можно ссылаться на Signal.red (красный сигнал), Signal.green (зеленый сигнал) и т.д. Этим избегается случайное присвоение неправильных значений переменным типа Signal.

Листинг 18.1. Спецификация очереди с помощью объявления интерфейса

```
interface Queue {
    public void put (Object o) ;
    public void remove (Object o) ;
    public int size ();
} //Queue
```

Листинг 18.2. Объявление класса Signal в языке Java

```
class Signal {
    static public final int red = 1 ;
    static public final int amber = 2 ;
    static public final int green = 3 ;
    public int sigState ;
}
```

18.1.3. Разработка безотказного ПО

Для разработки программного обеспечения с минимальным числом отказов необходимо иметь технологический процесс, который хорошо определен, опробован и содержит этапы контроля и аттестации ПО. Хорошо определенный процесс – это стандартизированный и документированный процесс. Опробованный процесс не зависит от индивидуальной интерпретации и решений разработчиков ПО. (Эти вопросы подробно рассматриваются в главе 25.)

Процесс разработки ПО должен включать в себя хорошо спланированный комплексный процесс тестирования, имеющий целью обнаружение системных ошибок. Процесс тестирования, минимизирующий количество системных ошибок, имеет несколько составляющих.

1. *Проверка требований*, цель которой состоит в обнаружении ошибок и других проблем в системной спецификации (см. главу 6). Большая доля отказов и сбоев в работе готового программного продукта обусловлена ошибками в спецификации требований.
2. *Управление требованиями.* В задачу управления требованиями, как указывалось в главе 6, входит контроль за изменениями в требованиях и отслеживание внесения соответствующих изменений (вызванных изменением требований) на всех этапах создания системы. Многие ошибки в готовых системах появляются в результате того, что изменения требований не были учтены при проектировании или реализации системы.
3. *Проверка моделей.* Это автоматический анализ системных моделей с помощью инструментальных CASE-средств, которые контролируют внутреннюю и внешнюю непротиворечивость этих моделей. Внутренняя непротиворечивость означает непротиворечивость отдельной модели системы; внешняя – непротиворечивость всех моделей системы (например, модели состояний и модели объектов).
4. *Проверка системной архитектуры и инспекция кода программ.* Как будет показано в главе 19, проверка архитектуры и кода программ базируется на технологических картах проверки программ и предназначена для обнаружения и устранения этих ошибок перед тестированием системы.
5. *Статический анализ.* Это автоматизированный метод анализа программ, предназначенный для нахождения потенциально ошибочных условий. Этот метод описывается в главе 19.
6. *Планирование и управление тестированием системы.* Необходимо разработать набор всесторонних тестов и сам процесс тестирования, гарантирующий полную проверку системы. Вопросы тестирования систем рассматриваются в главе 20.

18.2. Устойчивость к сбоям

Устойчивой к сбоям (отказоустойчивой) является такая система, которая может продолжить работу после проявления некоторых системных сбоев. Другими словами, в отказоустойчивых системах сбои не приводят к отказу системы. Устойчивость к сбоям необходима там, где отказ системы может стать причиной катастрофических последствий или где остановка работы системы приводит к большим экономическим потерям. Например, компьютерные системы самолета должны безотказно работать до тех пор, пока он не при-

землится; система управления воздушным движением должна быть работоспособной, пока самолеты в воздухе.

Можно предположить, что средства отказоустойчивости не должны включаться в систему, которая разрабатывается на основе методов минимизации ошибок. Если в системе нет ошибок, то, казалось бы, нет и причин для системных отказов. Но "без ошибок" не означает "безотказный". "Без ошибок" только означает, что программа соответствует своей спецификации. Спецификация требований сама может содержать ошибки или упущения, требования же могут базироваться на неправильных предположениях о системном окружении. И конечно, никогда нельзя утверждать, что система полностью лишена ошибок. В системах, к которым предъявляются высочайшие требования безотказности и работоспособности, явно необходима поддержка устойчивости к отказам.

В проблеме безотказности выделяют четыре аспекта.

1. *Обнаружение ошибок и сбоев.* Система должна обнаруживать "сбойные" состояния, которые могут привести к ее отказу.
2. *Локализация сбоев.* Определение той части системы, в которой возникли сбои.
3. *Восстановление системы.* После возникновения сбоя система должна вернуться в работоспособное состояние. Этого можно достичь исправлением сбойного состояния (прямое устранение ошибки) или возвращением системы к "безопасному" состоянию (ретроспективное устранение ошибки).
4. *Устранение причин сбоев.* Система модифицируется таким образом, чтобы сбои не возобновлялись. Во многих случаях сбои программного обеспечения проявляются кратковременно, поскольку возникают из-за специфической комбинации данных на входе системы. Так как нормальное функционирование системы восстанавливается сразу после устранения сбоя, нет необходимости в немедленном устранении причин сбоев. В этом состоит важное отличие сбоев программного обеспечения от неисправности аппаратных средств.

Существует два дополняющих друг друга подхода, которые используются для разработки ПО, устойчивого к сбоям.

1. *Безопасное программирование.* Это такой метод разработки программ, при котором программисты допускают, что в их программах могут быть необнаруженные ошибки или противоречия. В процессе изменения программы для проверки состояния системы включаются избыточные коды, чтобы гарантировать, что изменения непротиворечивы. Если противоречия обнаружены, от изменений отказываются или состояние восстанавливается до известного корректного состояния.
2. *Отказоустойчивые системные архитектуры.* Это архитектуры аппаратных и программных средств, обеспечивающие устойчивость к сбоям. Такие архитектуры включают резервирование аппаратных и программных средств и имеют блок анализа сбоев, который обнаруживает и устраняет ошибки. Этот подход к обеспечению отказоустойчивости рассматривается в разделе 18.3.

Безопасное программирование можно использовать при разработке любой системы, поскольку средства проверки и восстановления при сбоях следует использовать даже в тех программах, где появление ошибок маловероятно. Однако, прежде чем обсудить этот подход, рассмотрим обработку исключений, как необходимое средство обеспечения устойчивости к сбоям.

18.2.1. Обработка исключений

Исключением (или исключительной ситуацией) называется ситуация, когда во время выполнения программы возникают ошибки или непредвиденные события. Примерами исключительных ситуаций могут быть сбой в электропитании системы, попытка доступа к несуществующим элементам данных, числовые переполнения, исчезновение значащего разряда и т.д. Исключительные ситуации могут быть вызваны несовершенством программного обеспечения или оборудования. Когда возникает исключительная ситуация, она должна контролироваться и управляться системой. Это можно сделать непосредственно внутри самой программы либо путем передачи управления механизму обработки исключений.

В языках программирования, таких как C, чтобы обнаружить исключительную ситуацию и передать управление программе обработки исключительных ситуаций, используется оператор условного перехода if. Такому подходу сопутствуют две проблемы.

1. Исключения могут происходить в различных точках программы, и одно и то же исключение может произойти в разных местах. Это означает, что в программе должен быть контроль за большим количеством исключительных ситуаций. Это увеличивает размер программы, усложняет ее и делает более трудной для понимания, что повышает вероятность появления в ней ошибок.
2. Когда исключительная ситуация возникает в последовательности вложенных функций или вызывающих процедур, не существует простого способа передать ее от одной функции к другой, поскольку управление передается через последовательность процедур. Рассмотрите ситуацию, показанную на рис. 18.2, где функция A вызывает функцию B, которая, в свою очередь, вызывает функцию C. Если исключительная ситуация произойдет во время выполнения функции C, то это может сделать невозможным выполнение функции B. В этом случае необходим немедленный возврат от функции B к функции A с сообщением, что работа функции B завершилась неверно и что произошла исключительная ситуация.

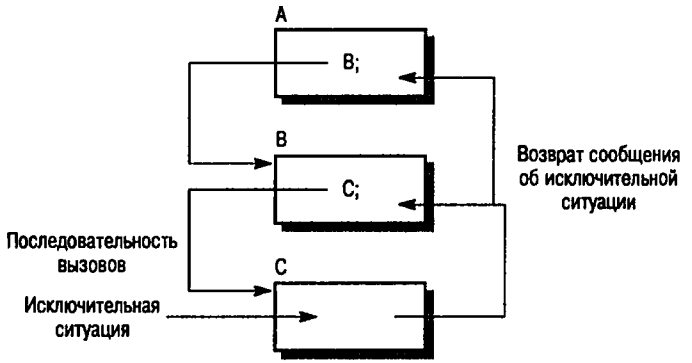


Рис. 18.2. Исключительная ситуация в последовательности вложенных функций

Если язык программирования имеет конструкции, поддерживающие обработку исключительных ситуаций, то дополнительные условные операторы для проверки исключительных ситуаций не понадобятся. Для этого язык программирования должен поддерживать специальный встроенный тип данных (часто называемый Exception (Исключение)) и

должны быть объявлены различные исключительные ситуации этого типа. Когда происходит исключительная ситуация, генерируется сигнал об исключительной ситуации и управление передается обработчику исключений. Подпрограмма обработчика определяет исключение и применяет соответствующие действия для его обработки.

В языках Ada, C++ и Java предусмотрены специальные средства обработки исключительных ситуаций. В языке Java новые типы исключительных ситуаций можно объявить путем расширения встроенного класса Exception. Для сообщения об исключительной ситуации в языке Java используется оператор throw (перемещать). Программа обработки исключительных ситуаций определяется ключевым словом catch (захват), за которым следует программный блок, обрабатывающий исключения.

Листинг 18.3 иллюстрирует обработку исключений в языке Java. Это часть программного обеспечения для системы инъекций инсулина, описанной в главе 16. Здесь программный блок управляет датчиком, определяющим величину содержания сахара в крови. В листинге 18.3 сначала объявляется класс исключений путем расширения встроенного класса объектов Exception. Далее приведен код обработки исключительной ситуации.

Листинг 18.3. Исключительные ситуации в языке Java

```
class SensorlFailureException extends Exception {
    SensorFailureException (String msg) {
        super (msg);
        Alarm.activate (msg);
    }
} //SensorFaifureException
class Sensor {
    int readVal () throws SensorFailureException {
    try {
        int theValue = DeviceIO.readInteger ();
        if (theValue < 0)
            throw new SensorFailureFxception ("Отказ датчика");
        return theValue;
    }
    catch (deviceIOException e)
        {
            throw new SensorFailureException ("Ошибка датчика");
        }
    } // readVal
} // Sensor
```

Класс Sensor (Датчик) обеспечивает метод readVal (чтение значения датчика), который содержит в объявлении оператор throw. Это означает, что исключение SensorFailureException (Исключение отказа датчика) можно передать из метода. Ключевое слово try (попытка) указывает, что исключение может быть передано в следующий блок программы. Исключение SensorFailureException передается, если значение, возвращаемое датчиком, меньше нуля. Метод DeviceIO.readInteger передает исключение deviceIOException (исключение ввода-вывода устройства) обработчику, расположенному после ключевого слова catch.

Средства обработки исключительных ситуаций в языках программирования можно использовать не только для управления системными сбоями. Они могут также применяться для обработки ошибок невыполнения условий, хотя таковые обычно происходят очень редко. Эта возможность показана в листинге 18.4. Здесь на языке Java реализовано термореле морозильной камеры для пищевых продуктов. Температура может изменяться от –18 до –40 градусов Цельсия.

Листинг 18.4. Исключительные ситуации управления морозильником

```
class FreezerController {
    Sensor tempSensor =  new Sensor ();
    Dial tempDial = new Dial () ;
    float freezerTemp = tempSensor.readVal ();
    final float dangerTemp= (float) -18.0 ;
    final long coolingTime= (long) 200000.0;
    public void run () throws InterruptedException {
    try {
        Pump.switchIt (Pump.on);
        do {
            if (freezerTemp > tempDial.setting ())
                if (Pump.status == Pump.off)
                {
                    Pump.switchIt (Pump.on);
                    Thread.sleep (coolingTime);
                }
            else
                if (Pump.status == Pump.on)
                    Pump.switchIt (Pump.off);
            if (freezerTemp > dangerTemp)
                throw new FreezerTooHotException ();
            freezerTemp = tempSensor.readVal ();
        }  while (true);
    } //блок try
    catch (FreezerTooHotException f)
    { Alarm.activate(); }
    catch(InterruptedException e)
    (
        System.out.println("Потоковое исключение");
        throw new InterruptedException();
    }
    }
} // FreezerController
```

Замороженные продукты начинают размораживаться, а бактерии становятся активными при температуре выше –18 градусов. Система управления поддерживает эту температуру путем включения и выключения насоса для холодильного агента, в зависимости от значения температурного датчика. Если нужная температура не поддерживается, блок управления посылает аварийный сигнал.

В приведенном листинге температура морозильной камеры определяется путем опроса объекта tempSensor (температура датчика), а необходимая температура – объекта tempDial (установка температуры). Объект Pump (насос) отвечает на сигналы, изменяя свое состояние. Если насос включился, система ожидает некоторое времени (вызывая метод Thread.sleep) для понижения температуры. Если она не понижается в достаточной мере, происходит переход к обработке исключительной ситуации FreezerTooHotException.

Обработчик исключений (помещенный в конец программы) фиксирует эту исключительную ситуацию и активизирует объект Alarm (Сигнал тревоги). Также предусмотрен обработчик исключения InterruptedException (исключение прерывания), которое может быть передано из метода Thread.sleep. Эта исключительная ситуация затем передается основному методу.

18.2.2. Обнаружение ошибок и сбоев

Языки программирования подобные Java и Ada имеют строгие описания типов. Это позволяет многие ошибки, которые служат причиной нарушения состояний и отказов системы, обнаружить во время компиляции. Компилятор может обнаружить нарушения правил описания типов. Проверка компилятора ограничена статическими величинами, но компилятор может также автоматически сгенерировать программу, которая выполняет, например, проверку присвоения значений массивам. Она проверит, что индексы массива не выйдут за указанные пределы и что значения массива имеют необходимый тип.

Первая ступень средств отказоустойчивости должна обнаружить, что сбой (ошибочное состояние системы) произошел или произойдет, если немедленно не будут выполнены некоторые действия. Обычно в такой ситуации управление передается в блок программы, которая может управлять обнаруженным сбоем.

Существует два типа обнаружения сбоев.

1. *Превентивное обнаружение ошибок.* В этом случае механизм обнаружения ошибок запускается прежде, чем произойдет изменение состояния. Если потенциальное ошибочное состояние обнаружено, то изменения состояния не происходит. Обычно система обнаружения ошибок обрабатывает исключительную ситуацию, которая определяет тип обнаруженной ошибки.
2. *Ретроспективное обнаружение ошибок.* В этом случае механизм обнаружения ошибок запускается после того, как произошел сбой. Если ошибка обнаружена, сообщается об исключительной ситуации и используется механизм исправления ошибки.

Превентивное обнаружение ошибок часто выполняется путем ввода ограничений, накладываемых на состояния системы, и контроля за ними при переходе от одного состояния к другому. Эта схема упрощается, если состояние системы определяется состоянием ряда объектов. Ограничения, которые применяются к отдельным объектам, можно проверять автоматически и изменять во время выполнения метода (ассоциированного с объектом), который может изменить состояние. Этот подход проиллюстрирован в листинге 18.5, который на языке Java описывает класс PositiveEven, реализующий тип положительных четных чисел.

Листинг 18.5. Описание класса PositiveEven

```
class PositiveEvenInteger {
    int val = 0;
    PositiveEvenInteger (int n) throws NumericException
    {
        if (n < 0 | n%2 == 1)
            throw new NumericException ();
        else
            val = n;
    } //PositiveEvenInteger
    public void assign (int n) throws NumericException
    {
        if (n < 0 | n%2 == 1)
            throw new NumericException ();
        else
            val = n;
```

```
    } //присвоение
    int toInteger ()
    {
        return val;
    } //для целых чисел
    boolean equals (PositiveEvenInteger n)
    {
        return (val == n.val);
    } //равенство
} //PositiveEven
```

В этом примере ограничение состоит в том, что данный тип применим только к целым числам (которые, естественно, являются положительными и четными). Если нарушается это ограничение, возникает исключительная ситуация NumericException, о чем сообщается выполняемому методу. Отметим, что исключительная ситуация не обрабатывается внутри класса PositiveEvenInteger. За обработку исключений всегда должен отвечать вызывающий метод.

Превентивное обнаружение ошибок в значительной степени позволяет избежать проблем аварийного устранения отказов. Но это приводит к существенному увеличению действий в каждом операторе, который может изменить состояние, поскольку ограничения необходимо проверить до того, как произойдет изменение состояния. В некоторых системах, где важны эффективность и производительность и где сравнительно просто восстановить состояние системы, может использоваться ретроспективное обнаружение ошибок. Кроме того, если ограничения накладываются на взаимосвязи между объектами, их нельзя проверить, контролируя ограничения только отдельного объекта.

Ретроспективное обнаружение ошибок всегда включает контроль ограничений состояния, но в данном случае исследуется состояние в целом и проверяются все переменные, определяющие состояния. Отдельные функции проверки можно реализовать на языке Java, используя следующий интерфейс:

```
interface CheckableObject {
    public boolean check ();
}
```

Контролируемые объекты – это реализации класса объектов, который использует этот интерфейс, поэтому каждый объект имеет функцию проверки. Каждый класс реализует собственную функцию проверки, которая определяет ограничения, применяемые к объектам именно этого класса. Этот подход проиллюстрирован в листинге 18.6, где функция проверки исследует ограничения, которым должны удовлетворять элементы массива. Ретроспективное обнаружение ошибок, применяемое к нескольким переменным состояния, показано в листинге 18.7.

18.2.3. Локализация ошибок и сбоев

Локализация ошибок и сбоев заключается в анализе состояния системы для определения масштаба ее разрушения вследствие сбоя. Во многих случаях этого можно избежать, определяя наличие ошибок прежде, чем произойдет окончательное изменение состояния. Если ошибка обнаружена, изменение состояния не допускается, что предупреждает повреждение системы. Но локализация ошибок и сбоев необходима, когда изменения состояния нельзя избежать или когда сбой является результатом последовательности отдельных правильных состояний, которые, тем не менее, приводят к сбойному состоянию системы.

Локализация ошибок и сбоев состоит не в исправлении ошибок, а в оценке того, какие части пространства состояний пострадали от сбоя. Это можно сделать с помощью некой "функции законности", которая проверяет, является ли данное состояние непротиворечивым ("законным"). Если состояния "незаконные", они каким-либо способом выделяются.

Пример локализации ошибок на языке Java приведен в листинге 18.6. Здесь структура данных RobustArray (устойчивый массив) является коллекцией объектов типа CheckableObject (проверяемый объект). Класс, реализующий тип CheckableObject, должен содержать метод check (проверка), который проверяет, удовлетворяет ли значение объекта некоторому ограничению. Это имеет смысл, поскольку детали проверки зависят от типа CheckableObject, а не от объекта RobustArray.

Листинг 18.6. Класс массивов с локализацией ошибок

```
class RobustArray {
    //Проверка: все ли объекты массива
    //удовлетворяют определенным ограничениям
    boolean [] checkState ;
    CheckableObject [] theRobustArray;
    RobustArray (CheckableObject [] theArray)
    {
        checkState = new boolean [theArray.length];
        theRobustArray = theArray;
    } //RobustArray
    public void assessDamage () throws ArrayDamagedException
    {
        boolean hasBeenDamaged = false;
        for (int i = 0 ; i < this.theRobustArray.length ; i++)
        {
            if (!theRobustArray [i].check())
                {
                    checkState [i] = true;
                    hasBeenDamaged = true;
                }
            else
                checkState [i] = false;
        }
        if (hasBeenDamaged)
            throw new ArrayDamagedException ();
    }//assessDamage
}//RobustArray
```

Метод assessDamage (оценка ущерба) класса RobustArray проверяет правильность каждого элемента массива. Если элемент массива не соответствует ограничениям, которые определены в функции check, он записывается в массив checkState. Этим определяется исключение ArrayDamagedException. Обработчик исключительной ситуации должен быть включен в вызывающий метод. При обработке исключения используется информация из checkState.

Среди других методов, используемых для обнаружения ошибок и локализации повреждений, выделим следующие.

1. Использование контрольных сумм и контроль разрядов в числовых данных.
2. Использование избыточных связей в структурах данных, которые содержат указатели.
3. Использование в параллельных системах контрольных таймеров.

При изменении данных и обмене ими для проверки программ можно использовать контрольную сумму числовых данных [121]. Контрольная сумма – это величина, которая рассчитывается на основе данных с помощью специальной математической функции. Эта функция должна дать единственное значение для группы данных, которая участвует в обмене. Контрольную сумму вычисляет отправитель данных и добавляет это значение к данным. Приемник данных применяет к данным ту же самую функцию и сравнивает полученное значение с контрольной суммой. Если они отличаются, значит, произошло некоторое нарушение целостности данных. Этот же механизм может использоваться для обнаружения вторжения в защищенные данные и преднамеренного их изменения.

Когда используются связанные структуры данных, их представление можно сделать избыточным путем включения обратных ссылок. Тогда для каждой прямой ссылки от A к B будет существовать обратная ссылка от B к A. Если также имеется счетчик числа элементов в структуре, можно проверить соответствие прямых и обратных ссылок и совпадение эталонного и вычисленного размеров структуры.

Если процессы имеют ограничения на время их выполнения, можно установить контрольный таймер. Он начинает действовать одновременно с процессом, определяя время его исполнения, и возвращается в исходное состояние после выполнения процесса. Он опрашивается блоком управления через постоянные интервалы времени. Если по каким-то причинам процесс не завершится, контрольный таймер не возвратится в исходное положение. В таком случае блок управления обиаруживает сбой и принимает меры для принудительного завершения процесса.

18.2.4. Восстановление системы

Это процесс изменения состояний системы для минимизации последствий сбоев. В этом случае система может продолжать функционировать, возможно с пониженной эффективностью. Прямое восстановление системы (при отсутствии дублирующих компонентов) – это попытка исправить поврежденное состояние системы. Обратное восстановление переводит систему из сбойного состояния к известному "правильному" состоянию.

Прямое восстановление системы обычно применяется в следующих ситуациях.

1. *Когда разрушены закодированные данные.* Использование методов кодирования, которые добавляют данным избыточность и позволяют не только определить, но и исправить ошибки.
2. *Когда разрушены связанные структуры.* Если в систему данных включены как прямые, так и обратные указатели, то структура может быть восстановлена в случае сохранения достаточного количества указателей. Эта методика часто используется для восстановления баз данных и файловых систем.

Обратное восстановление системы является более простым методом, который восстанавливает систему путем перевода ее в безопасное состояние. Большинство систем баз данных имеют средство обратного восстановления. Когда пользователь

начинает работу с базой данных, инициализируются транзакции. Изменения, сделанные в течение выполнения транзакции, не переносятся немедленно в базу данных. База данных изменяется только после окончания транзакций в том случае, если не возникло никаких проблем. Если при выполнении транзакции произошел сбой, база данных не изменяется.

Такой процесс выполнения транзакций позволяет восстановление при возникновении ошибок, поскольку изменения в базах данных не происходят до окончания выполнения транзакции. Но процесс выполнения транзакций не позволяет восстановление из состояния, которое было изменено. Введение контрольных точек – метод, который позволяет выйти из этой ситуации. Состояние системы периодически дублируется. Когда возникают проблемы, корректное состояние можно восстановить, воспользовавшись одной из этих копий.

Пример обратного восстановления на основе обработки исключительной ситуации показан в листинге 18.7, в котором приведен код на языке Java для обнаружения ошибок и обратного восстановления.

Листинг 18. 7. Процедура безопасной сортировки с обратным восстановлением

```
class SafeSort {
    static void sort int [] intarray, int order ) throws SortError
    {
        int [] copy = new int [intarray.length];
        //копирование исходного массива
        for (int i = 0 ; i < intarray.length ; i++)
            copy[i] = intarray[i];
        try {
            Sort.bubblesort (intarray, intarray.length, order);
            if (order == Sort.ascending)
                for (int i = 0 ; i <= intarray.length - 2 ; i++)
                    if (intarray [i] > intarray [i+1])
                        throw new SortError();
            else
                for (int  i = 0 ; i <= intarray.length - 2 ; i++)
                    if (intarray [i+1] > intarray [i])
                        throw new SortError ();
        } //блок try
        catch (SortError e)
        {
            for (int i = 0 ; i < intarray.length ; i++)
                intarray [i] = copy [i] ;
            throw new SortError ("Массив не отсортирован");
        } // catch
    } // сортировка
} // SafeSort
```

Метод копирует массив перед выполнением сортировки. В этом примере для простоты сортировка выполняется методом "пузырька", но, очевидно, можно использовать любой алгоритм сортировки. Если в алгоритме сортировки есть ошибки, то массив не будет отсортирован, что определяется точным порядком элементов в массиве. В этом случае возникает исключительная ситуация SortError (ошибка сортировки). Обработчик исключе-

ний не пробует устранить проблему, а восстанавливает первоначальные значения массива и повторно передает SortError, чтобы указать вызывающему методу, что сортировка не была успешно завершена. В этом случае ответственность за устранение ошибок перекладывается на вызывающий метод.

18.3. Отказоустойчивые архитектуры

Безопасное программирование — эффективный метод обеспечения отказоустойчивости. Он относительно прост и обычно не намного усложняет систему. Однако он не может эффективно справиться с ошибками системы, которые являются результатом взаимодействия оборудования и программного обеспечения. Кроме того, ошибки в системной спецификации или неправильная интерпретация требований могут привести к тому, что безопасное программирование не сможет спасти правильность программного кода. Поэтому для большинства критических систем, особенно со строгими требованиями к работоспособности, требуется системная архитектура, обеспечивающая устойчивость к сбоям. Примерами систем, которым необходим такой вид системной архитектуры, являются самолетные системы, которые должны сохранять работоспособность в течение всего времени полета, телекоммуникационные системы, а также разнообразные системы управления и контроля.

Для построения отказоустойчивого оборудования обычно требуется много лет. Используемая в большинстве случаев технология устойчивого к сбоям оборудования базируется на методе тройного модульного резервирования, когда модуль оборудования дублируется три (иногда больше) раза. Выходные данные каждого модуля сравниваются: если один из модулей выходит из строя и на его выходе данные не совпадают с выходными данными других модулей, эти данные игнорируются. Если невозможно сразу восстановить сбойный модуль, система автоматически переконфигурируется, исключая поврежденный модуль. Далее система продолжает функционировать с двумя работающими модулями (рис. 18.3).

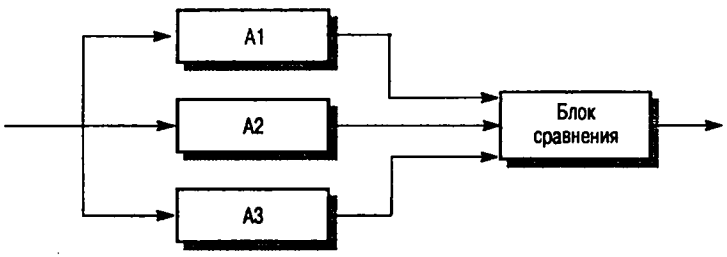


Рис. 18.3. Тройное модульное резервирование для обеспечения отказоустойчивости оборудования

Этот метод обеспечения отказоустойчивости предполагает, что большинство сбоев в работе оборудования являются результатом отказа компонентов, а не ошибок проектирования системы; при этом считается, что отказы компонентов проявляются независимо. Также предполагается, что все составляющие системы удовлетворяют спецификации требований. Вследствие этого вероятность одновременного отказа всех компонентов оборудования считается очень малой.

Конечно, компоненты могут иметь общую ошибку проектирования и тогда могут одновременно отказать. Вероятность такой ситуации можно уменьшить, если использовать

компоненты, которые удовлетворяют общим требованиям, но проектировались и разрабатывались разными командами разработчиков. В этом случае предполагается, что вероятность допустить одну и ту же проектную или производственную ошибку различными командами разработчиков очень мала.

Подобно тому как к аппаратным средствам предъявляются требования работоспособности и безотказности, аналогичные требования устойчивости к сбоям предъявляются и к программному обеспечению. Существует два сравнимых подхода к обеспечению отказоустойчивости ПО (рис. 18.4 и 18.5). Оба подхода заимствованы из моделей аппаратных средств, где вследствие избыточности неисправные компоненты можно временно исключить из системной конфигурации.

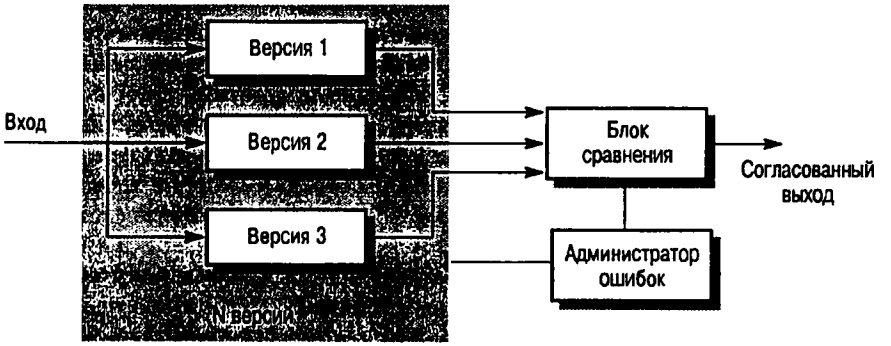


Рис. 18.4. N-вариантное программирование

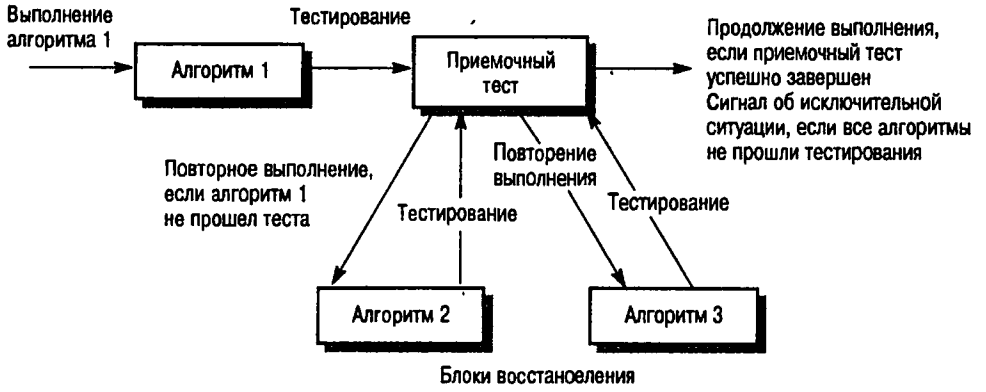


Рис. 18.5. Блоки восстановления

Опишем подходы к созданию отказоустойчивого программного обеспечения.

1. *N-вариантное программирование.* В соответствии с общей спецификацией различными командами разработчиков разрабатывается несколько версий программного обеспечения. Эти версии выполняются параллельно на отдельных компьютерах. Результаты их работы сравниваются с помощью системы согласования, результаты какой-либо версии, не совпадающие с двумя другими или не полученные вовремя, отвергаются. Это наиболее часто используемый подход к созданию отказоустойчивого программного обеспечения. Он используется в системах сигна-

лизации на железных дорогах, в авиационных системах и в системах защиты реакторов [14, 15, 29*].

2. *Блоки восстановления.* В этом методе каждый программный компонент содержит тест, проверяющий его работу. Компонент также имеет подпрограмму, которая повторяет вычисления, если тест обнаружил неправильное выполнение. Но повторные вычисления выполняются другим модулем компонента, который намеренно построен на другой интерпретации требований, предъявляемых к компоненту. Таким образом, компонент имеет несколько модулей (блоков восстановления), выполняющих одинаковые функции, но реализующих разные алгоритмы. Поскольку модули выполняются последовательно, в рамках данного подхода нет необходимости дублировать аппаратные средства. Метод блоков восстановления описан в работах [288, 289, 8*].

Оба описанных подхода используют несколько разных архитектур и программных реализаций. Когда для реализации одной и той же спецификации используется несколько различных подходов, естественно считать маловероятной ситуацию, когда различные версии ПО будут иметь одни и те же ошибки. Достичь различия между разными версиями ПО можно также следующими способами.

1. Включение в системную спецификацию требований использовать при проектировании различные подходы. Например, можно потребовать, чтобы одна команда разработчиков использовала объектно-ориентированное проектирование, а другая – функционально-ориентированное.
2. Включение в спецификацию требований, чтобы для реализации были использованы различные языки программирования. Например, в системе с тремя версиями ПО для написания разных версий можно использовать языки программирования Ada, C++ и Java.
3. Использовать при разработке системы различные инструментальные средства и среды разработки.
4. Можно потребовать использования конкретных алгоритмов для реализации определенных системных компонентов. Это, конечно, ограничивает свободу разработчиков и может стать причиной других проблем.

Каждая команда разработчиков должна работать со своей спецификацией (так называемой V-спецификацией), которую получают из общей спецификации системных требований [15]. Команды разработчиков каждой версии ПО должны работать изолированно, чтобы уменьшить вероятность появления в версиях одинаковых ошибок.

Многообразие версий, конечно, увеличит общую безотказность системы. Однако ряд экспериментов показывает: предположение о том, что различные команды разработчиков не сделают одинаковых ошибок, не всегда справедливо [200, 58, 214]. Различные коллективы разработчиков могут сделать одни и те же ошибки из-за одинаковых интерпретаций требований или вследствие того, что они независимо друг от друга используют одни и те же алгоритмы. Метод блоков восстановления уменьшает вероятность общих ошибок, поскольку блоки восстановления реализуют разные алгоритмы.

Слабость обоих методов отказоустойчивости состоит в том, что они основаны на предположении правильности системной спецификации. Однако во многих случаях

спецификации неполны или содержат ошибки. Один из путей уменьшения возможных ошибок в общей спецификации состоит в независимой разработке V-спецификаций и реализации их с помощью различных нотаций. Тогда один коллектив разработчиков может работать с формальной спецификацией, другой — с моделью системы, описывающей ее состояния, а третий — с требованиями на естественном языке. Это поможет избежать ошибок интерпретации требований, но не освободит от ошибок в самой спецификации.

18.4. Проектирование безопасных систем

Общее правило проектирования ПО, критического по обеспечению безопасности, основывается на сокрытии информации и простоте программного обеспечения. Те части системы, которые являются критическими, должны быть изолированы от других частей системы. Этого можно достигнуть с помощью абстракций данных и управления физическим разделением системы. Критическая часть программного обеспечения может выполняться на отдельном компьютере с минимальными связями с другими частями системы.

Программное обеспечение, критическое по обеспечению безопасности, должно быть простым настолько, насколько возможно. Потенциально подверженных ошибкам конструкций языков программирования, рассмотренных выше в главе, нужно избегать везде, где возможно. Они даже могут быть запрещены стандартом разработки критических систем. В некоторых случаях можно разработать требование, чтобы программы писались на подмножестве языка, в котором исключены ненадежные конструкции. Такие подмножества были разработаны для языков Modula-2, Ada и Pascal, и, вероятно, будет разработано безопасное подмножество языка Java.

Отказоустойчивое программное обеспечение должно использоваться только в системах, критических по обеспечению безопасности, когда состояния не защищены и безопасность системы зависит от работоспособности. Методы, повышающие устойчивость к отказам, усложняют создание и проверку программного обеспечения. Как упоминалось в предыдущем разделе, исследования показали, что использование N-вариантного программирования не всегда обеспечивает безотказность систем, поскольку при разработке ПО на основе одной спецификации, различные команды разработчиков могут делать одни и те же ошибки. Избыточность программного обеспечения не дает теоретически предсказанного увеличения системной безотказности. Кроме того, если спецификация некорректна, все версии ПО будут иметь одинаковые ошибки.

Это не означает, что N-вариантное программирование бесполезно. Оно может уменьшать абсолютное число отказов системы. Если предположить, что общие ошибки будут обнаружены в различных версиях системы, то число отказов будет уменьшено. Кроме того, считается, что N-вариантное программирование увеличивает доверие к безотказности системы [40].

Альтернативой использованию N-вариантного программирования, которое усложняет систему, является создание максимально простых (насколько это возможно) систем и выделение дополнительных ресурсов для проверки правильности системы. Простота программного обеспечения снижает вероятность ошибок. Это также сокращает обычно высокие затраты на проверку безопасности системы, поскольку только сравнительно небольшая часть ПО будет связана с обеспечением безопасности.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Функциональная надежность программ может быть достигнута путем исключения ошибок и включения средств отказоустойчивости, которые обеспечивают продолжение функционирования системы даже после того, как ошибки в системе приведут к сбою.
- Некоторые программные конструкции и методы, например операторы безусловного перехода, указатели, рекурсия, наследование и числа с плавающей запятой, по своей природе могут послужить причиной ошибок. Они не должны использоваться при разработке надежных систем.
- Программное обеспечение, устойчивое к сбоям, может продолжать функционирование, несмотря на ошибки, которые вызывают сбои системы.
- Существует четыре аспекта отказоустойчивости ПО, а именно: обнаружение ошибок и сбоев, локализация сбоев, восстановление системы и устранение причин сбоев.
- Безопасное программирование — это метод программирования, который объединяет проверку программ на наличие ошибок и их локализацию. Ошибки определяются ранее, чем они приведут к сбою системы.
- Метод безопасного программирования использует средства обработки исключительных ситуаций, существующие в таких языках программирования, как Java и C++.
- N-вариантное программирование и блоки восстановления являются методами построения отказоустойчивых архитектур, где устойчивость к сбоям обеспечивается дублированием программных и аппаратных средств.

Упражнения

18.1. Объясните, почему наследование является конструкцией, потенциально приводящей к ошибкам, и почему использование механизма наследования необходимо минимизировать при разработке критических объектно-ориентированных систем.

18.2. Если рекурсия — конструкция, приводящая к ошибкам, спроектируйте класс объектов, реализующий двоичные деревья и работу с ними без использования рекурсии.

18.3. Опишите три метода безопасного программирования, которые уменьшают вероятность того, что ошибки ПО приведут к сбоям системы.

18.4. Кратко опишите стратегии прямого и обратного восстановления систем. Почему обратное восстановление используется чаще, чем прямое? Приведите два класса систем, где может использоваться обратное восстановление.

18.5. Какие предварительные условия должны соблюдаться, прежде чем может быть выполнено прямое восстановление отказоустойчивой системы? Возможно ли прямое восстановление в интерактивных системах?

18.6. Спроектируйте абстрактный тип данных или объектный класс RobustList (устойчивый список), который реализует прямое восстановление в связанном списке. Для этого используйте прямые и обратные ссылки между соседними элементами списка.

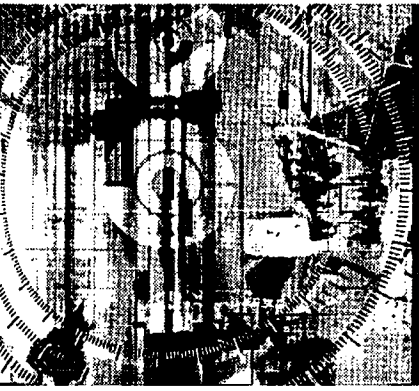
18.7. Опишите обстоятельства, когда при построении программных систем управления необходимо использовать отказоустойчивую архитектуру, и объясните почему.

18.8. Предложим, что управляющее программное обеспечение для лучевой терапии (используется в лечении больных раком пациентов) реализуется с помощью N-вариантного программирования. Прокомментируйте, правильное ли это решение или нет.

18.9. Укажите две причины, почему различные версии системы в N-вариантных системах могут дать одинаковый сбой.

18.10. Обсудите проблемы разработки и сопровождения "безостановочных" систем типа программного обеспечения телефонной станции. Как можно использовать при разработке таких систем исключительные ситуации?

18.11. Использование методов разработки безопасного программного обеспечения, обсуждавшихся в этой главе, очевидно, требует значительных дополнительных затрат. Какие дополнительные затраты можно оправдать, если за 15-летний срок службы системы будут сохранены 100 жизней? Были бы оправданы те же самые затраты, если будут сохранены 10 жизней? Какова цена жизни? Правомочно и этично ли в таких ситуациях проводить оценку затрат?



Часть II

Верификация и аттестация

19 Верификация и аттестация ПО

Цели

Цель настоящей главы — дать общее представление о верификации и аттестации программного обеспечения и познакомить с методами статической верификации. Прочитав эту главу, вы должны:

- знать отличие между верификацией и аттестацией ПО;
- познакомиться с инспектированием программ — эффективным методом выявления ошибок в программах;
- понять, почему статический анализ программ является одним из основных методов верификации;
- познакомиться с разработкой программ методом "чистая комната" и знать, почему этот метод оказывается эффективным.

Содержание

Верификацией и аттестацией называют процессы проверки и анализа, в ходе которых проверяется соответствие программного обеспечения своей спецификации и требованиям заказчиков. Верификация и аттестация охватывают полный жизненный цикл ПО – они начинаются на этапе анализа требований и завершаются проверкой программного кода на этапе тестирования готовой программной системы.

Верификация и аттестация не одно и то же, хотя их легко перепутать. Кратко различие между ними можно определить следующим образом [45]:

- верификация отвечает на вопрос, правильно ли создана система;
- аттестация отвечает на вопрос, правильно ли работает система.

Согласно этим определениям, верификация проверяет соответствие ПО системной спецификации, в частности функциональным и нефункциональным требованиям. Аттестация – более общий процесс. Во время аттестации необходимо убедиться, что программный продукт соответствует ожиданиям заказчика. Аттестация проводится после верификации, для того чтобы определить, насколько система соответствует не только спецификации, но и ожиданиям заказчика.

Как уже отмечалось в главе 6, на ранних этапах разработки ПО очень важна аттестация системных требований. В требованиях часто встречаются ошибки и упущения; в таких случаях конечный продукт, вероятно, не будет соответствовать ожиданиям заказчика. Но, конечно, аттестация требований не может выявить все проблемы в спецификации требований. Иногда недоработки и ошибки в требованиях обнаруживаются только после завершения реализации системы.

В процессах верификации и аттестации используются две основные методики проверки и анализа систем.

1. *Инспектирование ПО.* Анализ и проверка различных представлений системы, например документации спецификаций требований, архитектурных схем или исходного кода программ. Инспектирование выполняется на всех этапах процесса разработки программной системы. Параллельно с инспектированием может выполняться автоматический анализ исходного кода программ и соответствующих документов. Инспектирование и автоматический анализ – это статические методы верификации и аттестации, поскольку им не требуется исполняемая система.
2. *Тестирование ПО.* Запуск исполняемого кода с тестовыми данными и исследование выходных данных и рабочих характеристик программного продукта для проверки правильности работы системы. Тестирование – это динамический метод верификации и аттестации, так как применяется к исполняемой системе.

На рис. 19.1 показано место инспектирования и тестирования в процессе разработки ПО. Стрелки указывают на те этапы процесса разработки, на которых можно применять данные методы. Согласно этой схеме, инспектирование можно выполнять на всех этапах процесса разработки системы, а тестирование – в тех случаях, когда создан прототип или исполняемая программа.

К методам инспектирования относятся: инспектирование программ, автоматический анализ исходного кода и формальная верификация. Но статические методы могут проверить только соответствие программ спецификации, с их помощью невозможно проверить правильность функционирования системы. Кроме того, статическими методами нельзя проверить такие нефункциональные характеристики, как производительность и надежность. Поэтому для оценивания нефункциональных характеристик проводится тестирование системы.

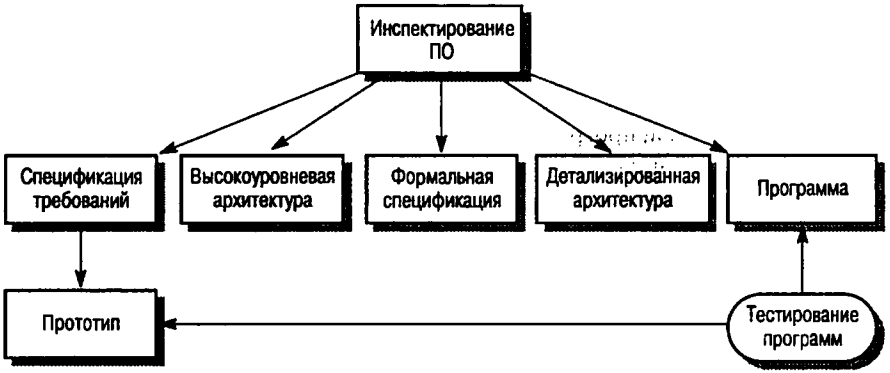


Рис. 19.1. Статическая и динамическая верификация и аттестация

В настоящее время, несмотря на широкое применение инспектирования ПО, преобладающим методом верификации и аттестации все еще остается тестирование. Тестирование – это проверка работы программ с данными, подобными реальным, которые будут обрабатываться в процессе эксплуатации системы. Наличие в программе дефектов и несоответствий требованиям обнаруживается путем исследования выходных данных и выявления среди них аномальных. Тестирование выполняется на этапе реализации системы (для проверки соответствия системы ожиданиям разработчиков) и после завершения ее реализации.

На разных этапах процесса разработки ПО применяют различные виды тестирования.

1. *Тестирование дефектов* проводится для обнаружения несоответствий между программой и ее спецификацией, которые обусловлены ошибками или дефектами в программах. Такие тесты разрабатываются для выявления ошибок в системе, а не для имитации ее работы. Данный вид тестирования рассматривается в главе 20.
2. *Статистическое тестирование* оценивает производительность и надежность программ, а также работу системы в различных режимах эксплуатации. Тесты разрабатываются так, чтобы имитировать реальную работу системы с реальными входными данными. Надежность функционирования системы оценивается по количеству сбоев, отмеченных в работе программ. Производительность оценивается по результатам измерения полного времени выполнения операций и времени отклика системы при обработке тестовых данных. Статистическое тестирование и оценивание надежности рассматриваются в главе 21.

Конечно, между этими методами не существует жестких, четко установленных границ. Во время тестирования дефектов испытатель может получить интуитивное представление о надежности ПО, а во время статистического тестирования есть возможность выявления программных дефектов.

Главная цель верификации и аттестации – удостовериться в том, что система "соответствует своему назначению". Соответствие программной системы своему назначению отнюдь не предполагает, что в ней совершенно не должно быть ошибок. Скорее, система должна достаточно хорошо соответствовать тем целям, для которых планировалась. Уровень необходимой *достоверности соответствия* зависит от назначения системы, ожиданий пользователей и условий на рынке программных продуктов.

1. *Назначение ПО.* Уровень достоверности соответствия зависит от того, насколько критическим является разрабатываемое программное обеспечение по тем или иным критериям. Например, уровень достоверности для систем, критическим по обеспечению безопасности, должен быть значительно выше аналогичного уровня достоверности для опытных образцов программных систем, разрабатываемых для демонстрации некоторых новых идей.
2. *Ожидания пользователей.* Следует с грустью отметить, что в настоящее время у большинства пользователей невысокие требования к программному обеспечению. Пользователи настолько привыкли к отказам, происходящим во время работы программ, что не удивляются этому. Они согласны терпеть сбои в работе системы, если преимущества ее использования компенсируют недостатки. Вместе с тем с начала 1990-х годов терпимость пользователей к отказам в работе программных систем постепенно снижается. В последнее время создание ненадежных систем стало практически неприемлемым, поэтому компаниям, занимающимся разработкой программных продуктов, необходимо все больше внимания уделять верификации и аттестации программного обеспечения.
3. *Условия рынка программных продуктов.* При оценке программной системы продавец должен знать конкурирующие системы, цену, которую покупатель согласен заплатить за систему, и назначенный срок выхода этой системы на рынок. Если у компании-разработчика несколько конкурентов, необходимо определить дату выхода системы на рынок до окончания полного тестирования и отладки, иначе первыми на рынке могут оказаться конкуренты. Если покупатели не желают приобретать ПО по высокой цене, возможно, они согласны терпеть большее количество отказов в работе системы. При определении расходов на процесс верификации и аттестации необходимо учитывать все эти факторы.

Как правило, в ходе верификации и аттестации в системе обнаруживаются ошибки. Для исправления ошибок в систему вносятся изменения. Этот *процесс отладки* обычно интегрирован с другими процессами верификации и аттестации. Вместе с тем тестирование (или более обобщенно – верификация и аттестация) и отладка являются разными процессами, которые имеют различные цели.

1. Верификация и аттестация – процесс обнаружения дефектов в программной системе.
2. Отладка – процесс локализации дефектов (ошибок) и их исправления (рис. 19.2).

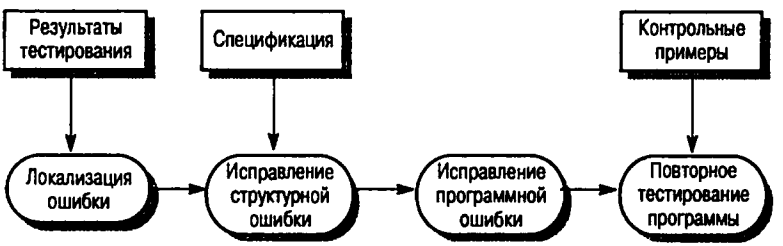


Рис. 19.2. Процесс отладки

Простых методов отладки программ не существует. Опытные отладчики обнаруживают ошибки путем сравнения шаблонов тестовых выходных данных с выходными данными тестируемых систем. Чтобы определить местоположение ошибки, необходимы знания о типах ошибок, шаблонах выходных данных, языке программирования и процессе программирова-

ния. Очень важны знания о процессе разработке ПО. Отладчикам известны наиболее распространенные ошибки программистов (например, связанные с пошаговым увеличением значения счетчика). Также учитываются ошибки, типичные для определенных языков программирования, например связанные с использованием указателей в языке С.

Определение местонахождения ошибок в программном коде не всегда простой процесс, поскольку ошибка необязательно находится возле того места в коде программы, где произошел сбой. Чтобы локализовать ошибки, программист-отладчик разрабатывает дополнительные программные тесты, которые помогают выявить источник ошибки в программе. Может возникнуть необходимость в ручной трассировке выполнения программы.

Интерактивные средства отладки являются частью набора средств поддержки языка, интегрированных с системой компиляции программного кода. Они обеспечивают специальную среду выполнения программ, посредством которой можно получить доступ к таблице идентификаторов, а оттуда к значениям переменных. Пользователи часто контролируют выполнение программы пошаговым способом, последовательно переходя от оператора к оператору. После выполнения каждого оператора проверяются значения переменных и выявляются возможные ошибки.

Обнаруженная в программе ошибка исправляется, после чего необходимо снова проверить программу. Для этого можно еще раз выполнить инспектирование программы или повторить предыдущее тестирование. Повторное тестирование используется для того, чтобы убедиться, что сделанные в программе изменения не внесли в систему новых ошибок, поскольку на практике высокий процент "исправления ошибок" либо не завершается полностью, либо вносит новые ошибки в программу.

В принципе во время повторного тестирования после каждого исправления необходимо еще раз запускать все тесты, однако на практике такой подход оказывается слишком дорогостоящим. Поэтому при планировании процесса тестирования определяются зависимости между частями системы и назначаются тесты для каждой части. Тогда можно трассировать программные элементы с помощью специальных контрольных примеров (контрольных данных), подобранных для этих элементов. Если результаты трассировки задокументированы, то для проверки измененного программного элемента и зависимых от него компонентов можно использовать только некоторое подмножество всего множества тестовых данных.

19.1. Планирование верификации и аттестации

Верификация и аттестация — дорогостоящий процесс. Для больших систем, например систем реального времени со сложными нефункциональными ограничениями, половина бюджета, выделенного на разработку системы, тратится на процесс верификации и аттестации. Поэтому очевидна необходимость тщательного планирования данного процесса.

Планирование верификации и аттестации, как один из этапов разработки программных систем, должно начинаться как можно раньше. На рис. 19.3 показана модель разработки ПО, учитывающая процесс планирования испытаний. Здесь планирование начинается еще на этапах создания спецификации и проектирования системы. Данную модель иногда называют V-моделью (чтобы увидеть букву V, необходимо повернуть рис. 19.3 на 90°). На этой схеме также показано разделение процесса верификации и аттестации на несколько этапов, причем на каждом этапе выполняются соответствующие тесты.

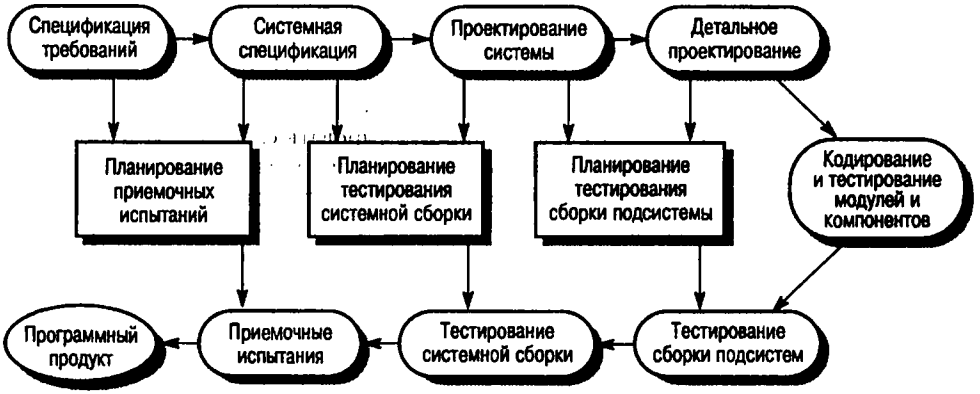


Рис. 19.3. Планирование испытаний в процессе разработки и тестирования

В процессе планирования верификации и аттестации необходимо определить соотношение между статическими и динамическими методами проверки системы, определить стандарты и процедуры инспектирования и тестирования ПО, утвердить технологическую карту проверок программ (см. раздел 19.2) и составить план тестирования программ. Чему уделить больше внимания — инспектированию или тестированию, зависит от типа разрабатываемой системы и опыта организации. Чем более крнтична система, тем больше внимания необходимо уделить статическим методам верификации.

Врезка 19.1. Структура плана испытаний ПО

Процесс тестирования

Описание основных этапов процесса тестирования.

Возможность отслеживания требований

Тестирование следует спланировать таким образом, чтобы протестировать все требования в отдельности.

Тестируемые элементы

Следует определить все "выходные" продукты процесса разработки ПО, которые необходимо тестировать.

График тестирования

Составляется временной график тестирования и распределение ресурсов согласно этому графику. Очевидно, что график тестирования привязан к более общему графику разработки проекта.

Процедуры записи тестов

Недостаточно только проводить тесты — результаты тестирования необходимо систематически записывать, чтобы потом можно было исследовать процесс тестирования и проверить правильность его выполнения.

Аппаратные и программные требования

В этом разделе определяются все необходимые для тестирования инструментальные программные и аппаратные средства.

Ограничения

В этом разделе следует попытаться предвидеть все неблагоприятные факторы, влияющие на процесс тестирования, например нехватку персонала.

В плане верификации и аттестации основное внимание уделяется стандартам процесса тестирования, а не описанию конкретных тестов. Этот план предназначен не только для руководства, он в основном предназначен специалистам, занимающимся разработкой и тестированием систем. План дает возможность техническому персоналу получить полную картину испытаний системы и в этом контексте спланировать свою работу. Кроме того, план предоставляет информацию менеджерам, отвечающим за то, чтобы у группы тестирования были все необходимые аппаратные и программные средства.

Основные компоненты плана испытаний ПО перечислены во врезке 19.1. Хорошее описание подобных планов и их взаимосвязи с более общими планами обеспечения качества представлено в работе [118].

Подобно другим планам, план испытаний не является неизменным документом. Его следует регулярно пересматривать, так как тестирование зависит от процесса реализации системы. Например, если реализация какой-либо части системы не завершена, то невозможно провести тестирование сборки системы. Поэтому план необходимо периодически пересматривать, чтобы сотрудников, занятых тестированием, можно было использовать на других работах.

19.2. Инспектирование программных систем

Системное тестирование программ требует разработки огромного количества тестов, их выполнения и проверки. Это значит, что данный процесс достаточно трудоемкий и дорогостоящий. Каждый тест позволяет обнаруживать одну, а в лучшем случае несколько ошибок в программе. Причина такого положения заключается в том, что сбои в работе, происходящие из-за ошибок в системе, часто приводят к разрушению данных. Поэтому трудно сказать, какое количество ошибок "ответствеино" за сбой в системе.

Инспектирование программ не требует их исполнения, поэтому данный метод можно использовать до завершения полной реализации программ. Во время инспектирования проверяется исходное представление системы. Это может быть модель системы, спецификация или программа, написанная на языке высокого уровня. Для обнаружения ошибок используется знание разрабатываемой системы и семантика ее исходного представления. Каждую ошибку можно рассматривать отдельно, не обращая внимания на то, как она влияет на поведение системы.

Доказано, что инспектирование является эффективным методом обнаружения ошибок. Также немаловажно, что инспектирование значительно дешевле экстенсивного тестирования программ. В экспериментах, описанных в работе [27], сравнивалась эффективность инспектирования и тестирования. Инспектирование программного кода оказалось более эффективным и менее дорогостоящим, чем тестирование. Такие же выводы сделаны в работе [129].

В статье [112] утверждается, что более 60% ошибок в программах можно обнаружить с помощью неформального исследования (инспектирования) программ. При более формальном подходе, использующем математические методы, в программе можно обнаружить более 90% всех ошибок [239]. Такая проверка используется в процессе разработки систем методом "чистая комната", который рассматривается в разделе 19.4. Процесс инспектирования также может оценить другие качественные характеристики систем: соответствие стандартам, переносимость и удобство сопровождения. Качественные характеристики систем рассматриваются в главе 24.

В системных компонентах и подсистемах выявление ошибок путем просмотра и инспектирования обычно более эффективно, чем с помощью тестирования, по двум причинам.

1. За один сеанс инспектирования можно выявить множество разнообразных программных дефектов. Недостатком тестирования является то, что обычно за один сеанс тестирования можно обнаружить только одну ошибку, поскольку ошибки могут привести к отказу системы или их эффекты могут накладываться друг на друга.
2. Инспектирование использует знания о предметной области и языке программирования. Специалист, проводящий инспектирование, должен знать типы ошибок, присущие конкретным языкам программирования и приложениям определенного типа. Поэтому в ходе анализа программ есть возможность сосредоточиться только на конкретных типах ошибок.

Конечно, инспектирование не может полностью заменить тестирование. Инспектирование лучше использовать как начальный процесс верификации для обнаружения большей части программных дефектов. Путем инспектированиия проверяют соответствие ПО ее спецификации, однако таким способом нельзя проверить динамическое поведение системы. Более того, нерационально инспектировать законченные системы, собранные из нескольких подсистем. На этом уровне возможно только тестирование. Тестирование также необходимо для оценки надежности и производительности, проверки пользовательского интерфейса и соответствия системы требованиям заказчика.

Инспектирование и тестирование не являются конкурирующими методами верификации и аттестации. Каждому из них присущи свои преимущества и недостатки, поэтому в процессе верификации и аттестации их следует использовать совместно. Одним из наиболее эффективных методов инспектирования является применение контрольных примеров [129]. В этом случае можно обнаружить программные дефекты и разработать более эффективные методы тестирования системы.

Иногда при инспектировании в организации, разрабатывающей традиционное программное обеспечение, возникают трудности. Разработчики, имеющие опыт тестирования программ, неохотно соглашаются с тем, что инспектирование оказывается более эффективным методом выявления ошибок, чем тестирование. Менеджеры относятся к этим технологиям с недоверием, потому что внедрение инспектирования на этапах проектирования и разработки требует дополнительных расходов. Инспектирование всегда требует расходов, причем на начальном этапе разработки ПО, а конечная экономия средств вследствие применения инспектирования достигается только благодаря опыту проводящих его спецналистов.

В этой главе рассматривается инспектирование программ, т.е. исходный код проверяется на наличие ошибок. Однако метод инспектирования можно также использовать для верификации любых текстовых докумеитов, созданных в процессе разработки ПО. Метод инспектирования можно применять к спецификации требований, для детализированного определения системной архитектуры, при разработке структур данных, планировании тестирования и в процессе создания системной документации.

19.2.1. Инспектирование программ

Инспектирование программ — это просмотр и проверка программ с целью обнаружения в них ошибок. Идея формализованного процесса проверки (инспекции) программ впервые сформулирована IBM в 1970-х годах и описана в работах [111, 112]. В настоящее время данный метод верификации программ получил широкое применение. На базе исходиого метода инспектирования разработано много других вариантов инспектирования программ [129]. Но все они основываются на базовой идее метода инспектирования, согласно которому группа специалистов выполняет тщательный построчный просмотр и анализ исходного кода программы.

Основное отличие инспектирования от других видов оценивания качества программ состоит в том, что главная его цель – обнаружение дефектов, а не исследование общих проблем проекта. Дефектами являются либо ошибки в программе, либо несоответствие программы организационным или проектным стандартам. В противоположность инспектированию другие методы анализа программ основное внимание уделяют организационным вопросам, временному графику работ, затратам, сравнению с промежуточными контрольными элементами или оценке соответствия ПО определенным целям организации-разработчика.

Процесс инспектирования – это формализованный процесс, выполняемый небольшой группой специалистов, состоящей не более чем из четырех человек. Члены группы системно анализируют программу и определяют возможные дефекты. Согласно исходной концепции метода инспектирования члены группы должны выполнять следующие роли: автора, рецензента, инспектора и координатора. Рецензент "озвучивает" программный код, инспектор проверяет код с помощью тестов, координатор отвечает за организацию процесса.

По мере накопления опыта инспектирования в организациях могут появляться другие предложения по распределению ролей в группе. В ходе обсуждения результатов использования инспектирования, внедренного в процесс разработки программ в компании Hewlett-Packard, в статье [136] предлагается шесть ролей (табл. 19.1). Одно лицо может исполнять несколько ролей, поэтому количество членов в группе инспектирования может варьироваться.

Таблица 19.1. Роли в процессе инспектирования

Роль	Описание
Автор или владелец	Программист или разработчик, который отвечает за создание программы или документа, а также несет ответственность за исправление дефектов, обнаруженных в процессе инспектирования
Инспектор	Находит ошибки, упущения и противоречия в программах и документах; может также указать на более общие проблемы, находящиеся вне сферы действия инспекционной группы
Рецензент	Излагает код или документ на собрании инспекционной группы
Секретарь	Записывает результаты собрания инспекционной группы
Председатель или координатор	Управляет и организует процесс инспектирования. Докладывает о результатах инспектирования руководству компании
Руководитель группы	Занимается совершенствованием процесса инспектнрования, обновлениями технологических карт, разработкой стандартов и т.п.

Как показано в статье [136], роль рецензента необязательна. В этом случае исходный процесс инспектирования, в котором рецензирование программы является важной составляющей, соответственно изменяется. Такой же вывод содержится в работе [129].

Для начала процесса инспектирования программы необходимы следующие условия.

1. Наличие точной спецификации кода, предназначенного для инспектирования. Без полной спецификации невозможно обнаружить дефекты в проверяемом программном компоненте.
2. Члены инспекционной группы должны хорошо знать стандарты разработки.
3. В распоряжении группы должна была синтаксически корректная последняя версия программы. Нет никакого смысла рассматривать код, который "почти завершен".

На рис. 19.4 показан общий процесс инспектирования. Он адаптирован к требованиям организаций, использующих инспектирование программ.

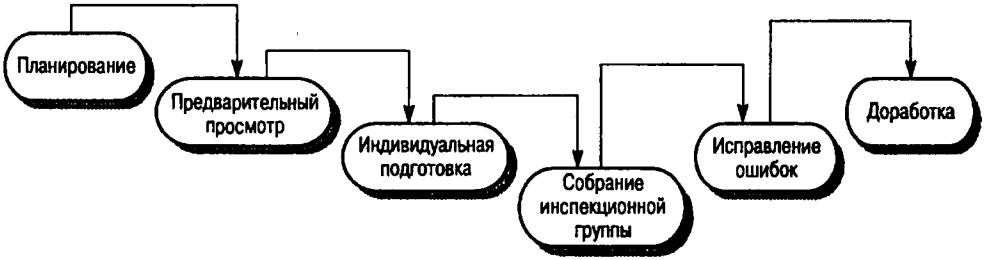


Рис. 19.4. Процесс инспектирования

Координатор составляет план инспектирований, подбирает инспекционную группу, организует собрание и убеждается, что программа и ее спецификация закончены. Программа, предназначенная для инспектирования, передается на рассмотрение инспекционной группе, где автор программы описывает ее назначение (этап предварительного просмотра). После этого следует этап индивидуальной подготовки, на котором каждый член инспекционной группы изучает программу и ее спецификацию и выявляет дефекты в программном коде.

Сам процесс инспектирования должен быть относительно коротким (не более двух часов) и сосредоточенным исключительно на выявлении дефектов, аномалий и несоответствий стандартам. Инспекционная группа не должна предлагать способы исправления обнаруженных дефектов или рекомендовать какие-либо изменения в других программных компонентах.

После инспектирования автор изменяет программу, исправляя обнаруженные ошибки. На этапе доработки координатор принимает решение о том, необходимо ли повторно проводить инспектирование. Если повторного инспектирования не требуется, все обнаруженные дефекты документально фиксируются и документ с результатами инспектирования утверждается председателем.

Процесс инспектирования должен проводиться с учетом технологической карты, описывающей возможные ошибки программирования. Карта разрабатывается квалифицированными специалистами и регулярно обновляется по мере накопления опыта в процессе инспектирования. Для разных языков программирования составляются разные технологические карты.

Технологические карты, составленные для разных языков программирования, различаются между собой, поскольку учитывают возможности проверки, которую обеспечивают компиляторы языков. Например, компилятор языка Ada проверяет количество параметров функций, а компилятор языка C — нет. Ошибки, которые можно выявить в процессе инспектирования, перечислены в табл. 19.2. Подчеркнем, что каждая организация должна разрабатывать собственные технологические карты для инспектирования, которые бы основывались на стандартах и опыте данной организации и обновлялись по мере обнаружения новых типов программных дефектов [129].

В процессе инспектирования организация накапливает определенный опыт, поэтому результаты инспектирования можно использовать для улучшения всего процесса разработки ПО. В ходе инспектирования выполняется анализ обнаруженных дефектов. Группа инспектирования и авторы инспектируемого кода определяют причины возникновения дефектов. Чтобы подобные дефекты не возникали в будущих системах, необходимо по возможности устранить причины возникновения дефектов, что означает внесение изменений в процесс разработки программных систем.

Таблица 19.2. Ошибки, выявляемые при инспектировании

Класс ошибок	Вопросы, помогающие выявлять ошибки
Ошибки данных	Все ли переменные в программе инициализированы до начала использования их значений?
	Все ли константы именованы?
	Равна ли верхняя граница массива его размеру или на единицу меньше этого размера?
	Какой разделитель используется для разделения символьных строк?
	Возможно ли переполнение буфера?
Ошибки управления	Выполняются ли условия для каждого условного оператора?
	Все ли циклы завершаются?
	Правильно ли в составных операторах расставлены скобки?
	Все ли выборы выполняются в операторах выбора?
Ошибки ввода-вывода	Используются ли в программе входные переменные?
	Всем ли выходным переменным перед выводом присваиваются значения?
	Могут ли какие-нибудь входные данные привести к нарушению системных данных?
Ошибки интерфейса	Все ли вызовы процедур и функций содержат правильное количество параметров?
	Согласованы ли типы формальных и фактических параметров?
	В правильном ли порядке расположены параметры?
	Если компоненты обращаются к разделяемой памяти, имеют ли они такую же модель структуры разделяемой памяти?
Ошибки управления памятью	Если связанная структура данных изменяется, правильно ли переопределяются все связи?
	Если используется динамическая память, правильно ли она распределяется?
	Происходит ли перераспределение памяти после того, как она больше не используется?
Ошибки управления исключениями	Все ли возможные ошибки рассмотрены в условиях, определяющих исключительные ситуации?

Количество кода, проверяемого за определенное время, зависит от опыта группы инспектирования, языка программирования и предметной области приложения. На основе опыта проведения инспектирования в компании IBM сделаны следующие оценки.

1. На этапе предварительного просмотра за один час можно просмотреть приблизительно 500 операторов исходного кода.
2. Во время индивидуальной подготовки за один час можно проверить примерно 125 операторов исходного кода.
3. На собрании за один час можно проверить от 90 до 125 операторов.

Эти цифры также подтверждаются данными, полученными во время проведения инспектирования компании AT&T [22].

Принято считать, что максимальная длительность инспектирования не должна превышать двух часов, поскольку потом эффективность обнаружения дефектов снижается. Поэтому в процессе разработки ПО инспектирование относительно малых системных компонентов должно выполняться достаточно часто.

Если команда инспектирования состоит из четырех человек, на проверку 100 строк кода требуется примерно один человеко-день. Считается, что сам процесс инспектирования занимает около часа, плюс каждый член команды тратит 1–2 часа на подготовку к инспектированию. Расходы на тестирование также существенно зависят от количества ошибок в программе. Но, с другой стороны, на инспектирование программы требуется вполовину меньше затрат, чем на эквивалентное тестирование программ.

Обеспечение инспектирования ПО требует квалифицированного управления и "правильного" отношения к результатам его проведения. Инспектирование — открытый процесс обнаружения ошибок, когда ошибки, допущенные отдельным программистом, неизбежно выявляются и становятся известны всей группе программистов. Менеджеры должны четко разграничивать инспектирование программного кода и оценку кадров. При оценке профессиональных качеств специалистов ни в коем случае нельзя учитывать ошибки, обнаруженные в процессе инспектирования. Руководителям инспекционных групп необходимо пройти тщательную подготовку, чтобы грамотно управлять процессом и совершенствовать культуру отношений, которая гарантировала бы поддержку в процессе обнаружения ошибок и отсутствие каких-либо обвинений в связи с этими ошибками.

19.3. Автоматический статический анализ программ

Статические анализаторы программ — это инструментальные программные средства, которые сканируют исходный текст программы и выявляют возможные ошибки и противоречия. Для анализаторов не требуется исполняемая программа. Они выполняют синтаксический разбор текста программы и опознают различные типы операторов. Таким образом, с помощью анализаторов можно проверить, правильно ли составлены операторы, сделать выводы относительно потока управления в программе и во многих случаях вычислить множество значений данных, используемых программой. Анализаторы дополняют средства обнаружения ошибок, предоставляемых компилятором языка.

Цель автоматического статического анализа — привлечь внимание проверяющего к аномалиям в программе, например к переменным, которые используются без инициализации или совсем не используются, или к данным, значения которых превышают заданное, и т.п. В табл. 19.3 перечислены типы ошибок, которые можно выявить с помощью статического анализа. Автоматический статический анализ лучше всего применять вместе с инспектированием ПО, так как он предоставляет дополнительную информацию инспекционной группе.

Таблица 19.3. Ошибки, обнаруживаемые статическим анализом

Тип ошибки	Описание ошибки
Ошибки данных	Переменные используются до их инициализации; переменные определены, но нигде не используются; переменным дважды присваиваются значения, однако между этими присвоениями они нигде не используются; выход за границы массива; переменные не определены
Ошибки управления	Неиспользуемый код; безусловные переходы в циклах
Ошибки ввода-вывода	Переменные выводятся дважды без промежуточного присвоения
Ошибки интерфейса	Неправильный тип параметра; неправильное количество параметров; результаты функции не используются; есть невызываемые процедуры и функции
Ошибки управления памятью	Ошибки в использовании указателей

Статический анализ состоит из нескольких этапов.

1. *Анализ потока управления.* На этом этапе идентифицируются и выделяются циклы, их точки входа и выхода, а также неиспользуемый код (это код, окруженный безусловными операторами перехода, или код одной из ветвей условного оператора, условие перехода к которой никогда не будет истинным).
2. *Анализ использования данных.* На этом этапе проверяется использование переменных в программе. Анализ позволяет обнаружить переменные, которые используются без предварительной инициализации, переменные, которые описаны дважды без промежуточного присвоения, а также объявленные, но нигде не используемые переменные. На этом этапе также можно выявить условные операторы с избыточными условиями. Это такие условия, значения которых никогда не изменяются: они либо всегда истинны, либо всегда ложны.
3. *Анализ интерфейса.* На этом этапе проверяется согласованность различных частей программы, правильность объявления процедур и их использования. Данный этап оказывается лишним, если используется язык со строгим контролем типов, например Java, так как подобный анализ выполняет компилятор этого языка. Анализ интерфейса помогает выявить ошибки в программах, написанных на языках со слабым контролем типов, например FORTRAN или C. В процессе анализа интерфейса можно также выявить объявленные функции и процедуры, которые нигде не вызываются, и функции, результаты которых не используются.
4. *Анализ потоков данных.* На этом этапе анализа определяются зависимости между исходными (входными) и результирующими (выходными) переменными. Хотя такой анализ не выявляет конкретных ошибок, он дает полный список значений, используемых в программе, благодаря чему легче обнаружить ошибочный вывод данных. На этом этапе также можно явно определить условия, которые влияют на значения переменных.
5. *Анализ ветвей программы.* На этом этапе семантического анализа определяются все ветви программы и выделяются операторы, исполняемые в каждой ветви. Анализ ветвей программы существенно помогает разобраться в управлении программой и позволяет проанализировать каждую ветвь отдельно.

Анализ потока данных и анализ ветвей генерируют огромное количество информации. Эта информация не выявляет конкретных ошибок, а представляет программу в разных аспектах. Из-за огромного количества генерируемой информации эти этапы статического анализа иногда исключают из процесса анализа и используют только на ранних стадиях для обнаружения аномалий в разрабатываемой программе.

Статические анализаторы особенно полезны в тех случаях, когда используются языки программирования, подобные C. В языке C нет строгого контроля типов, и потому проверка, осуществляемая компилятором языка C, ограниченна. В этом случае средствами статического анализа можно автоматически выявить широкий спектр ошибок программирования. Данный анализ особенно важен при разработке критических систем. В этом случае статический анализ позволяет значительно сократить расходы на тестирование.

В системах Unix и Linux есть статический анализатор LINT для программ, написанных на C. Он обеспечивает статическую проверку, эквивалентную проверке компилятором в языках со строгим контролем типов, например Java. В листинге 19.1 представлен образец результата проверки программы с помощью анализатора LINT. Первая команда анализатора просматривает программу. В программе определена функция printarray с одним параметром, которая затем вызывает ее с тремя параметрами. Переменные i и c определены, однако значения им нигде не присваиваются. Возвращаемое функцией значение также нигде не используется.

Листинг 19.1. Статический анализ, выполненный анализатором LINT

```
138% more lint_ex.c

#include <stdio.h>
printarray (Anarray)
int Anarray;
{
printf("%d",Anarray);
}
main()
{
    int Anarray[5]; int i; char c;
    printarray(Anarray,i,c);
    printarray(Anarray);
}
139% cc lint_ex.c
140% lint lint_ex.c

lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args.lint_ex.c(4) ::lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) ::lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) ::lint_ex.c(11)
printf returns value which is always ignored
```

В строке 139 выполняется компиляция C-программы, в результате которой компилятор не обнаружил ни одной ошибки. В следующей за ней строке следует вызов анализатора LINT, который находит ошибки и предоставляет отчет.

Статический анализатор выявил, что используемые скалярные переменные c и i не инициализированы, функция printarray вызывается с другим количеством аргументов, чем указано при ее определении. Также обнаружено непоследовательное использование пер-

вого аргумента в функции printarray, кроме того, анализатор обнаружил, что значение функции нигде не используется.

Анализ с помощью инструментальных средств не может заменить инспектирования, так как существуют такие типы ошибок, которые невозможно выявить с помощью статического анализа. Например, анализаторы могут обнаружить необъявленные переменные, однако они не в состоянии определить неправильные присвоения. В языках со слабым контролем типов, например C, статические анализаторы могут определить функции с неверным количеством и типом аргументов, однако не способны распознать ситуации, когда в функции пропущен неверный аргумент правильного типа.

Конечно, для таких языков, как C, статический анализ является эффективным методом обнаружения ошибок. Но в современных языках программирования, например Java, из языка удалены конструкции, способствующие появлению многих ошибок. Все переменные должны быть объявлены, отсутствуют операторы безусловного перехода, вследствие чего маловероятно случайное создание неиспользуемого кода, и осуществляется автоматическое управление памятью. Такой подход к устранению ошибок более эффективен для повышения надежности программ, чем любые методы обнаружения ошибок. Поэтому для Java-программ использовать автоматический статический анализ нерентабельно.

19.4. Метод "чистая комната"

При разработке ПО методом "чистая комната" (cleanroom) для устранения дефектов используется процесс строгого инспектирования [239, 75, 219, 284]. Цель данного метода — создание ПО без дефектов. Название "чистая комната" взято по аналогии с производством кристаллов полупроводников, где выращивание кристаллов без дефектов происходит в сверхчистой атмосфере (чистых комнатах). Я описываю этот метод в данной главе, поскольку согласно ему в процессе разработки ПО при проверке соответствия системных компонентов спецификациям тестирование заменяется инспектированием.

На рис. 19.5 представлена модель процесса разработки ПО методом "чистая комната", построенная на основе описания, приведенного в работе [219].

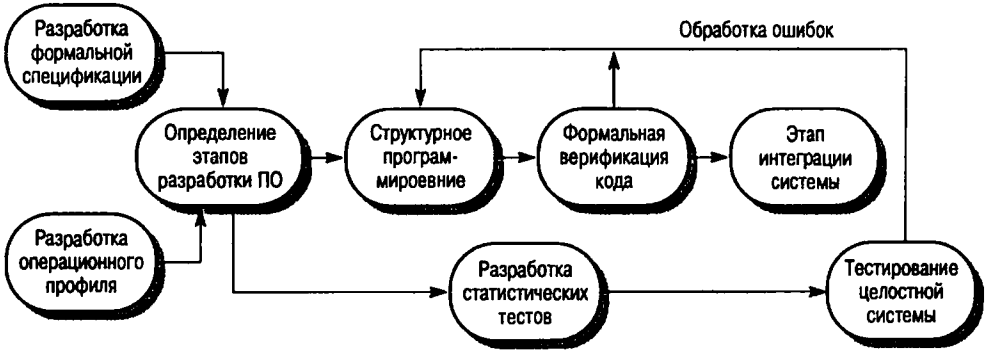


Рис. 19.5. Процесс разработки ПО методом "чистая комната"

В разработке ПО методом "чистая комната" можно выделить пять ключевых моментов.

1. *Формальная спецификация.* Для создаваемой системы разрабатывается формальная спецификация. Для записи спецификации используется модель состояний, в которой отображены отклики системы на стимулы.

2. *Пошаговая разработка.* Разработка ПО разбивается на несколько этапов, которые выполняются и проверяются методом "чистая комната" независимо друг от друга. Этапы определяются совместно с заказчиком на ранних стадиях процесса создания программного продукта.
3. *Структурное программирование.* Используется только ограниченное количество управляющих конструкций и абстракций данных. Процесс разработки программы — это процедура поэтапной детализации спецификации.
4. *Статическая верификация.* Разрабатываемое ПО проверяется статическим методом строгого инспектирования ПО. Для модулей или отдельных элементов тестирование кода не проводится.
5. *Статистическое тестирование системы.* На каждом шаге разработки проводится тестирование статистическими методами, позволяющими оценить надежность программной системы. Статистические методы рассматриваются в главе 21. Как показано на рис. 19.5, статистические тесты базируются на операционном профиле, который разрабатывается параллельно созданию спецификации системы.

Пошаговая разработка ПО, схема которой показана на рис. 19.6, описана в главе 3. Выполнение каждого этапа определяется пользователями. На каждом отдельном этапе получается вполне работоспособная система, но с ограниченными возможностями. Пользователи возвращают отчеты о функционировании системы вместе с предложениями необходимых изменений. Пошаговая разработка ПО позволяет уменьшить количество ошибок, возникающих из-за изменений требований заказчика.

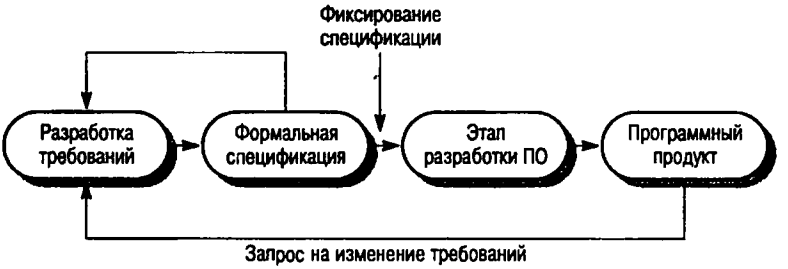


Рис. 19.6. Процесс пошаговой разработки

Если спецификация определена как единое целое, изменения в требованиях заказчика (которые неизбежны) влекут за собой изменения в спецификации и в процессе разработки. В этом случае спецификация и системная архитектура должны постоянно пересматриваться. В методе пошаговой разработки спецификация на каждом шаге фиксирована, хотя требования по изменению других частей системы принимаются. Каждый шаг разработки завершается готовым программным продуктом.

На первых этапах пошаговой разработки ПО методом "чистая комната" реализуются наиболее критические для заказчика системные функции. Менее важные системные функции добавляются на последующих этапах. Таким образом, у заказчика есть возможность проверить и испытать систему (ее основные функции) до окончательного завершения разработки. Если возникают проблемы с требованиями, заказчик сообщает об этом группе разработчиков и запрашивает новую версию продукта.

Таким образом, наиболее важные для заказчика системные функции проверяются наибольшее количество раз. По мере добавления в систему новые функции комбинируются с уже имеющимися, и интегрированная система тестируется. Поэтому те части системы,

которые созданы на первых этапах разработки, на каждом из последующих этапов проверяются еще раз с помощью других контрольных тестов.

Процесс разработки ПО методом "чистая комната" планируется таким образом, чтобы обеспечить строгое инспектирование программ. Спецификация системы представлена моделью состояний, которая через ряд последовательных моделей постепенно преобразуется в исполняемую программу. Этот подход к разработке ПО описан в главе 3. Инспектирование программ дополняются строгими математическими доказательствами согласованности и корректности преобразований.

Обычно разработкой больших систем методом "чистая комната" занимаются три группы разработчиков.

1. *Группа спецификации.* Отвечает за разработку и поддержку системной спецификации. Этой группой создаются спецификации пользовательских требований и формальные спецификации для верификации системы. В некоторых случаях, например после окончания разработки спецификации, эта группа может присоединиться к группе разработки.
2. *Группа разработки.* Занимается разработкой и проверкой ПО. При проверке используется структурированный формальный подход, основанный на инспектировании кода, подкрепленный доказательством правильности работы системы.
3. *Группа сертификации.* Занимается разработкой статистических тестов, применяемых после окончания разработки ПО. Все тесты основаны на использовании формальной спецификации. Контрольные тесты разрабатываются параллельно с созданием системы и используются для сертификации надежности ПО.

В результате использования метода "чистая комната" готовый программный продукт содержит крайне мало ошибок и его стоимость меньше, чем у разработанного традиционными методами. В работе [75] описано несколько успешных проектов, разработанных методом "чистая комната", с неизменно низким процентом ошибок в разработанных системах. Расходы на эти проекты сравнимы с расходами на проекты, которые разрабатывались с использованием традиционных методов.

В процессе разработки методом "чистая комната" оказывается рентабельной статическая проверка. Огромное количество дефектов обнаруживается еще до исполнения программы и исправляется в процессе разработки ПО. В статье [219] утверждается, что во время тестирования проектов, которые разрабатывались с использованием метода "чистая комната", в среднем обнаруживается только 2,3 дефекта на тысячу строк исходного кода. В целом расходы на разработку не увеличиваются, так как сокращаются расходы на тестирование и исправление ошибок в разрабатываемой программной системе.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Верификация и аттестация программных систем — разные процессы. Цель верификации — показать, что программа соответствует своей спецификации. Цель аттестации — показать, что программа работает именно так, как нужно пользователю.
- План тестирования ПО должен содержать описание тестируемых элементов, график проведения тестирования, описания процедур управления процессом тестирования, требования к аппаратному и программному обеспечению и описание проблем, которые могут возникнуть в процессе тестирования.

- Статические методы обнаружения ошибок проверяют и анализируют исходный программный код. Их следует использовать вместе с тестированием программ как часть процесса верификации и аттестации.
- Инспектирование программ является эффективным методом обнаружения ошибок в программе. Цель инспектирования — определить местоположение ошибок в программах. Процесс инспектирования должен проводиться в соответствии с технологической картой дефектов.
- Системная проверка кода программы проводится небольшой группой. Членами группы являются: руководитель группы (или координатор), автор (создатель) кода, рецензент, представляющий код во время инспектирования, и инспектор, проверяющий код с помощью различных тестов.
- Статические анализаторы — это программные инструментальные средства, которые анализируют исходный программный код. С их помощью можно выявить такие ошибки, как неиспользуемые разделы кода и необъявленные переменные.
- Разработка ПО методом "чистая комната" основана на статических методах проверки программ и статистическом тестировании для сертификации надежности системы. Метод успешно используется в процессе разработки систем с высоким уровнем надежности.

Упражнения

19.1. Обсудите различия между верификацией и аттестацией и объясните, почему аттестация является более сложным процессом.

19.2. Объясните, почему не нужно устранять все дефекты в программе перед ее поставкой заказчику. До каких пор следует тестировать программу, чтобы удостовериться, что она соответствует своему назначению?

19.3. Объясните, почему инспектирование программы является эффективным методом обнаружения в ней ошибок. Какие типы ошибок нельзя обнаружить методом инспектирования?

19.4. Разработайте технологическую карту наиболее распространенных ошибок (не синтаксических), которые не обнаруживаются компилятором, но которые можно выявить при инспектировании программы. При составлении таблицы используйте свои знания Java, C++, C или других языков программирования.

19.5. Составьте список вопросов, ответы на которые позволяют во время проведения статического анализа обнаружить ошибки в программах, написанных на языках Java, Ada и C++. Сравните свой список со списком, представленным в табл. 19.2.

19.6. Составьте отчет, в котором бы приводились преимущества метода "чистая комната", а также связанные с ним расходы и риски.

19.7. Менеджер решил для оценки специалистов в качестве исходных данных воспользоваться отчетами о результатах инспектирования программ. В отчетах содержится информация о том, кто совершил и кто обнаружил ошибки в программе. Этичны ли действия менеджера? Этично ли заранее проинформировать персонал об этом? Как это решение может повлиять на процесс инспектирования?

19.8. Один из подходов, широко используемых при тестировании ПО, состоит в тестировании системы до тех пор, пока не будут израсходованы все средства, выделенные на тестирование. Затем система передается заказчикам. Обсудите этичность такого подхода.

20 Тестирование программного обеспечения

Цели

Цель настоящей главы — познакомить с методами тестирования программного обеспечения, которые используются для обнаружения ошибок и дефектов в программах. Прочитав эту главу, вы должны:

- знать, какие методы тестирования используются для выявления программных ошибок;
- познакомиться с основными принципами тестирования интерфейсов;
- знать особенности покомпонентного тестирования и тестирования процесса сборки объектно-ориентированных систем;
- познакомиться с CASE-средствами, применяемыми для тестирования.

Содержание

В главе 3 рассматривалась общая схема процесса тестирования. Он начинается с тестирования отдельных программных модулей, например процедур и объектов. Затем модули компонуются в подсистемы и потом в систему, при этом проводится тестирование взаимодействий между модулями. Наконец, после сборки системы, заказчик может провести серию приемочных тестов, во время которых проверяется соответствие системы ее спецификации.

На рис. 20.1 показана схема двухэтапного процесса тестирования. На этапе покомпонентного тестирования проверяются отдельные компоненты. Это могут быть функции, наборы методов, собранные в один модуль, или объекты. На этапе тестирования сборки эти компоненты интегрируются в подсистемы или законченную систему. На этом этапе основное внимание уделяется тестированию взаимодействий между компонентами, а также показателям функциональности и производительности системы как единого целого. Но, конечно, на этапе тестирования сборки также могут обнаруживаться ошибки в отдельных компонентах, не замеченные на этапе покомпонентного тестирования.

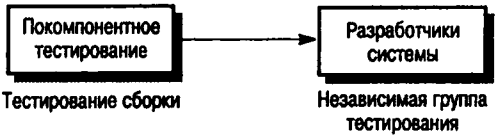


Рис. 20.1. Этапы тестирования ПО

При планировании процесса верификации и аттестации ПО менеджеры проекта должны определить, кто будет отвечать за разные этапы тестирования. Во многих случаях за тестирование своих программ (модулей или объектов) несут ответственность программисты. За следующий этап отвечает группа системной интеграции (сборки), которая интегрирует отдельные программные модули (возможно, полученные от разных разработчиков) в единую систему и тестирует эту систему в целом.

Для критических систем процесс тестирования должен быть более формальным. Такая формализация предполагает, что за все этапы тестирования отвечают независимые испытатели, все тесты разрабатываются отдельно и во время тестирования ведутся подробные записи. Чтобы протестировать критические системы, независимая группа разрабатывает тесты, исходя из спецификации каждого системного компонента.

При разработке некритических, "обычных" систем подробные спецификации для каждого системного компонента, как правило, не создаются. Определяются только интерфейсы компонентов, причем за проектирование, разработку и тестирование этих компонентов несут ответственность отдельные программисты или группы программистов. Таким образом, тестирование компонентов, как правило, основывается только на понимании разработчиками того, что должен делать компонент.

Тестирование сборки должно основываться на имеющейся спецификации системы. При составлении плана тестирования обычно используется спецификация системных требований или спецификация пользовательских требований (см. главу 3). Тестированием сборки всегда занимается независимая группа.

Во многих книгах, посвященных тестированию программного обеспечения, например [34, 197, 276, 22*], описывается процесс тестирования программных систем, реализующих функциональную модель ПО, но не рассматривается отдельно тестирование объектно-ориентированных систем. В контексте тестирования между объектно-ориентированными и функционально-ориентированными системами имеется ряд отличий.

1. В функционально-ориентированных системах существует четко определенное различие между основными программными элементами (функциями) и совокупностью этих элементов (модулями). В объектно-ориентированных системах этого нет. Объекты могут быть простыми элементами, например списком, или сложными, например такими, как объект метеорологической станции из главы 12, состоящий из ряда других объектов.
2. В объектно-ориентированных системах, как правило, нет такой четкой иерархии объектов, как в функционально-ориентированных системах. Поэтому такие методы интеграции систем, как нисходящая или восходящая сборка (см. раздел 20.2), часто не подходят для объектно-ориентированных систем.

Таким образом, в объектно-ориентированных системах между тестированием компонентов и тестированием сборки нет четких границ. В таких системах процесс тестирования является продолжением процесса разработки, где основной системной структурой являются объекты. Несмотря на то что бо́льшая часть методов тестирования подходит для систем любых видов, для тестирования объектно-ориентированных систем необходимы специальные методы. Такие методы рассмотрены в разделе 20.3.

20.1. Тестирование дефектов

Целью тестирования дефектов является выявление в программной системе скрытых дефектов до того, как она будет сдана заказчику. Тестирование дефектов противоположно аттестации, в ходе которой проверяется соответствие системы своей спецификации. Во время аттестации система должна корректно работать со всеми заданными тестовыми данными. При тестировании дефектов запускается такой тест, который вызывает *некорректную* работу программы и, следовательно, выявляет дефект. Обратите внимание на эту важную особенность: тестирование дефектов демонстрирует *наличие*, а не отсутствие дефектов в программе.

Общая модель процесса тестирования дефектов показана на рис. 20.2. Тестовые сценарии — это спецификации входных тестовых данных и ожидаемых выходных данных плюс описание процедуры тестирования. Тестовые данные иногда генерируются автоматически. Автоматическая генерация тестовых сценариев невозможна, поскольку результаты проведения теста не всегда можно предсказать заранее.

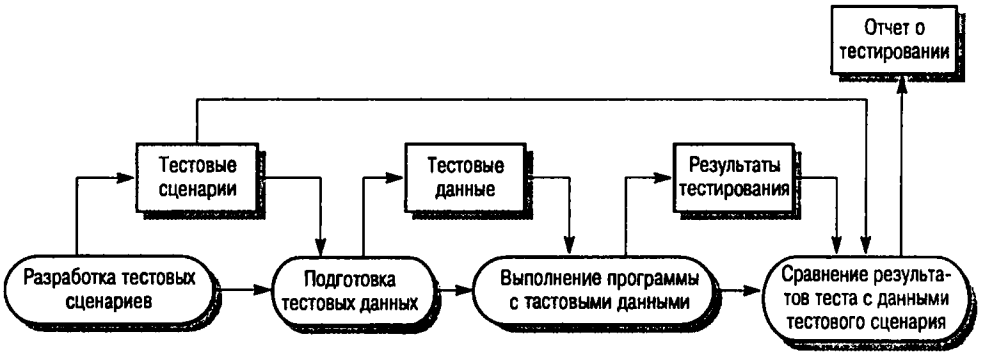


Рис. 20.2. Процесс тестирования дефектов

Полное тестирование, когда проверяются все возможные последовательности выполнения программы, нереально. Поэтому тестирование должно базироваться на некотором подмножестве всевозможных тестовых сценариев. Существуют различные методики выбора этого подмножества. Например, тестовые сценарии могут предусмотреть выполнение всех операторов в программе по меньшей мере один раз. Альтернативная методика отбора тестовых сценариев базируется на опыте использования подобных систем, в этом случае тестированию подвергаются только определенные средства и функции работающей системы, например следующие.

1. Все системные функции, доступные через меню.
2. Комбинации функций, доступные через меню (например, сложное форматирование текста).
3. Если в системе предполагается ввод пользователем каких-либо входных данных, тестируются функции с правильным и неправильным вводом данных.

Из опыта тестирования (и эксплуатации) больших программных продуктов, таких, как текстовые процессоры или электронные таблицы, вытекает, что необычные комбинации функций иногда могут вызывать ошибки, но наиболее часто используемые функции всегда работают правильно.

20.1.1. Тестирование методом черного ящика

Функциональное тестирование, или тестирование методом черного ящика базируется на том, что все тесты основываются на спецификации системы или ее компонентов. Система представляется как "черный ящик", поведение которого можно определить только посредством изучения ее входных и соответствующих выходных данных. Другое название этого метода — *функциональное тестирование* — связано с тем, что испытатель проверяет не реализацию ПО, а только его выполняемые функции.

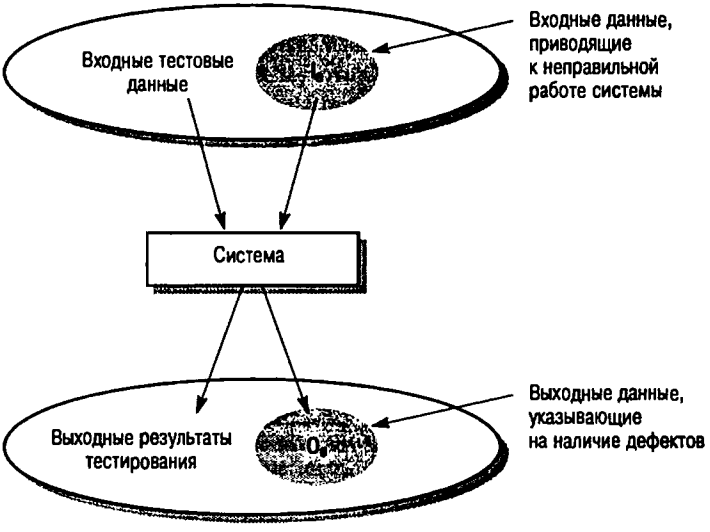


Рис. 20.3. Тестирование методом черного ящика

На рис. 20.3 показана модель системы, тестируемая методом черного ящика. Этот метод также применим к системам, организованным в виде набора функций или объектов. Испытатель подставляет в компонент или систему входные данные и исследует соответствующие выходные данные. Если выходные данные не совпадают с предсказанными, значит, во время тестирования ПО *успешно* обнаружена ошибка (дефект).

Основная задача испытателя — подобрать такие входные данные, чтобы среди них с высокой вероятностью присутствовали элементы множества I_e. Во многих случаях выбор тестовых данных основывается на предварительном опыте испытателя. Однако дополнительно к этим эвристическим знаниям можно также использовать систематический метод выбора входных данных, обсуждаемый в следующем разделе.

20.1.2. Области эквивалентности

Входные данные программ часто можно разбить на несколько классов. Входные данные, принадлежащие одному классу, имеют общие свойства, например это положительные числа, отрицательные числа, строки без пробелов и т.п. Обычно для всех данных из какого-либо класса поведение программы одинаково (эквивалентно). Из-за этого такие классы данных иногда называют областями эквивалентности [34]. Один из систематических методов обнаружения дефектов состоит в определении всех областей эквивалентности, обрабатываемых программой. Контрольные тесты разрабатываются так, чтобы входные и выходные данные лежали в пределах этих областей.

На рис. 20.4 каждая область эквивалентности изображена в виде эллипса. Области эквивалентности входных данных — это множества данных, все элементы которых обрабатываются одинаково. Области эквивалентности выходных данных — это данные на выходе программы, имеющие общие свойства, которые позволяют считать их отдельным классом. Корректные и некорректные входные данные также образуют две области эквивалентности.

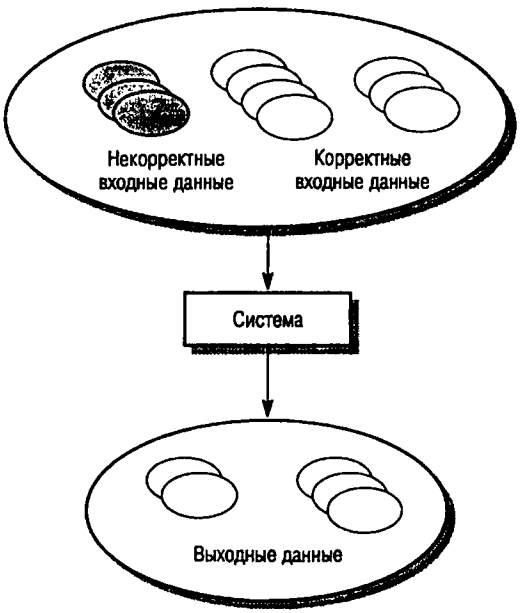


Рис. 20.4. Области эквивалентности

После определения областей эквивалентности для каждой из них подбираются тестовые данные. При выборе тестовых данных можно руководствоваться следующим полезным правилом: для тестов выбираются данные, расположенные на границе области эквивалентности, и отдельно данные, лежащие внутри этой области. Основная причина такого выбора данных заключается в следующем. В процессе разработки системы разработчики и программисты используют для тестов типичные значения входных данных, находящиеся внутри области эквивалентности. Граничные значения часто нетипичны (например, нулевое значение обрабатывается не так, как неотрицательные числа) и потому игнорируются программистами. Хотя чаще всего ошибки в программе возникают именно при обработке подобных нетипичных значений.

Области эквивалентности определяются на основании программной спецификации или документации пользователя и опыта испытателя, выбирающего классы значений входных данных, пригодные для обнаружения дефектов. Пусть, например, в спецификации программы указано, что в программу могут вводиться от 4 до 10 целых пятизначных чисел. Области эквивалентности и возможные значения тестовых входных данных для этого примера показаны на рис. 20.5.

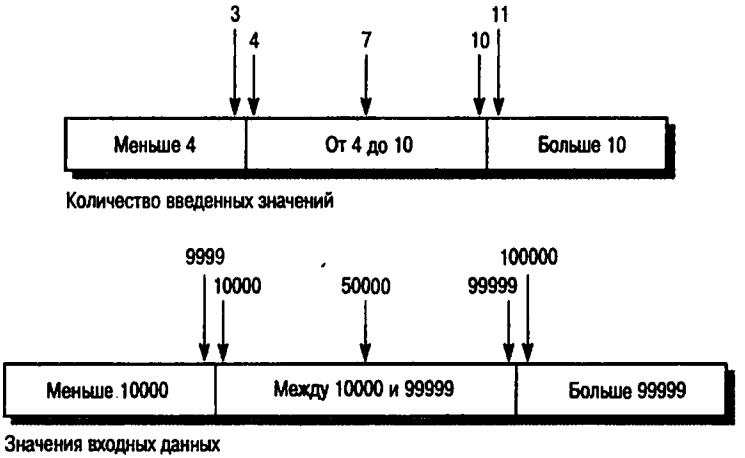


Рис. 20.5. Области эквивалентности

Покажем получение тестовых данных на примере спецификации упрощенной программы Seach (поиск), которая выполняет поиск заданного элемента key (ключ) в последовательности элементов. Программа возвращает номер позиции этого элемента в последовательности. Спецификация программы, представленная во врезке 20.1, содержит предусловие и постусловие. Предусловие указывает, что программа поиска не работает с пустыми последовательностями. Постусловие определяет, что если элемент, равный ключу, есть в последовательности, то переменная Found принимает значение true (истина). Индекс L обозначает позицию ключевого элемента в последовательности. Если элемент, равный ключу, в последовательности отсутствует, то этот индекс не определен.

Врезка 20.1. Спецификация программы поиска

```
Процедура: Seach (Key: ELEM; T: SEQ of ELEM;
        Found: in out BOOLEAN; L: in out ELEM_INDEX);
```

```
Предусловие
    -- в последовательности должен быть хотя бы один элемент
    T'FIRST<= T'LAST
Постусловие
    -- если элемент обнаружен под номером L
    (Found and T(L) = Key)
или
    -- если элемента нет в последовательности
    (not Found and
    not (exists i, T'FIRST >= i <= T'LAST, T(i) = Key ))
```

Согласно данной спецификации, можно определить две очевидные области эквивалентности:

- последовательности входных данных, содержащие ключевой элемент (Found = true);
- последовательности входных данных, не содержащие ключевого элемента (Found = false).

При определении областей эквивалентности руководствуются различными правилами. Вот несколько правил выбора тестирующих последовательностей.

1. Тестирующая последовательность может состоять из одного элемента. Обычно считается, что последовательности состоят из нескольких элементов, и программисты иногда закладывают такое представление в свои программы. Следовательно, если ввести последовательность из одного элемента, программа может сработать неправильно.
2. Следует использовать в разных тестах различные последовательности, содержащие разное количество элементов. Это уменьшает вероятность того, что программа, имеющая дефекты, случайно выдаст правильные результаты в силу некоторых случайных свойств входных данных.
3. Следует использовать тестирующие последовательности, в которых ключевой элемент является первым, средним и последним элементом последовательности. Такой метод помогает выявить проблемы на границах областей эквивалентности.

Исходя из этих правил, можно определить еще две области эквивалентности входных данных для программы Seach.

- Входная последовательность состоит из одного элемента.
- Во входной последовательности больше одного элемента.

Эти области комбинируются с определенными ранее областями эквивалентности, в результате будут получены области эквивалентности, представленные в табл. 20.1.

Таблица 20.1. Области эквивалентности для программы поиска

Последовательность	Ключевой элемент
Один элемент	Есть в последовательности
Один элемент	Нет в последовательности
Несколько элементов	Первый элемент последовательности
Несколько элементов	Последний элемент последовательности
Несколько элементов	Средний элемент последовательности
Несколько элементов	Нет в последовательности

Входная последовательиость (T)	Key	Выходные данные (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

В табл. 20.1 также представлен набор возможных тестовых данных, взятых из этих областей. Если ключевого элемента нет в последовательности, значение L не определено. При подборе тестовых данных применялось правило выбора последовательностей, согласно которому в разных тестах следует использовать последовательности разных размеров.

Множество вводимых значений, используемых для тестирования программы поиска, не является полным. Например, в работе программы может произойти сбой, если входная последовательность содержит элементы 1, 2, 3 или 4. Однако разумно предположить, что если не обнаружены дефекты при обработке одного элемента какого-либо класса эквивалентности, то тесты с любыми другими элементами этого класса также не выявят дефектов. Конечно, это не означает, что в программе отсутствуют дефекты. Возможно, не все области эквивалентности определены или определены неверно, или неправильно подобраны тестовые данные.

Здесь намеренно не рассматриваются тесты, которые проверяют порядок и тип используемых параметров. Возможные ошибки в использовании параметров лучше всего может выявить инспектирование программ или автоматический статический анализ. По этой же причине при тестировании не проверяется непредвиденное искажение данных на выходе программного компонента. Проблемы такого типа можно выявить во время инспектирования программ, которое рассматривается в главе 19.

20.1.3. Структурное тестирование

Метод структурного тестирования (рис. 20.6) предполагает создание тестов на основе структуры системы и ее реализации. Такой подход иногда называют тестированием методом "белого ящика", "стеклянного ящика" или "прозрачного ящика", чтобы отличать его от тестирования методом черного ящика.

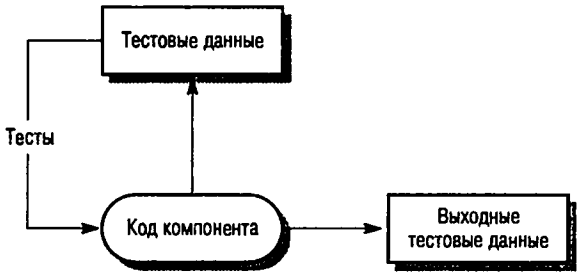


Рис.20.6. Структурное тестирование

Как правило, структурное тестирование применяется к относительно небольшим программным элементам, например к подпрограммам или методам, ассоциированным с объектами. При таком подходе испытатель анализирует программный код и для получения тестовых данных использует знания о структуре компонента. Например, из анализа кода можно определить, сколько контрольных тестов нужно выполнить для того, чтобы в процессе тестирования все операторы выполнились по крайней мере один раз.

Знание алгоритма, используемого при реализации некоторой функции, можно применять для определения областей эквивалентности. В качестве примера возьмем спецификацию программы поиска (см. врезку 20.1), реализованную на языке Java в виде процедуры бинарного поиска (листинг 20.1). Здесь реализованы более строгие предусловия. Последовательность представлена в виде массива, массив должен быть упорядоченным, значение нижней границы массива должно быть меньше значения верхней границы.

Листинг 20.1. Процедура бинарного поиска

```
class BinSearch {
//Реализация функции бинарного поиска;
//на входе: упорядоченный массив объектов и ключевой элемент key
//Возвращает объект с двумя атрибутами:
//index - значение индекса массива
//found - логическая переменная,
//показывает, есть или нет ключевой элемент в массиве
//Если в массиве нет элемента, совпадающего с key, key = -1
    public static void search (int key, int [] elemArray, Result r)
    {
        int bottom = 0;
        int top = elemArray.length - 1;
        int mid;
        r.found = false; r.index = -1;
        while ( bottom <= top )
        {
            mid = (top + bottom) / 2;
            if (elemArray [mid] == key)
            {
                r.index = mid;
                r.found = true;
                return;
            } //часть if
            else
            {
                if (elemArray[mid] < key)
                    bottom = mid + 1;
```

```
                else
                    top = mid - 1;
            }
        } //цикл while
    } // поиск
} //BinSearch
```

Из текста программы видно, что во время ее выполнения область поиска разделяется на три части, каждая из которых является областью эквивалентности (рис. 20.7). При проверке программы в качестве тестовых данных необходимо взять последовательности с ключевыми элементами, расположенными на границах этих областей.

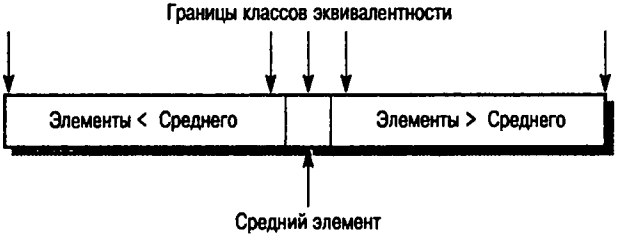


Рис. 20.7. Классы эквивалентности для бинарного поиска

Тестовые данные, представленные в табл. 20.7, необходимо изменить, поскольку элементы входного массива должны быть отсортированы в возрастающем порядке. Кроме того, следует добавить тестовые данные, где ключевой элемент расположен возле среднего элемента массива. Полученное множество тестовых данных для программы бинарного поиска представлено в табл. 20.2.

Таблица 20.2. Тестовые данные для программы бинарного поиска

Входной массив (T)	Ключ (Key)	Результат (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

20.1.4. Тестирование ветвей

Это метод структурного тестирования, при котором проверяются все независимо выполняемые ветви компонента или программы. Если выполняются все независимые ветви, то и все операторы должны выполняться по крайней мере один раз. Более того, все условные операторы тестируются как с истинными, так и с ложными значениями условий. В объектно-ориентированных системах тестирование ветвей используется для тестирования методов, ассоциированных с объектами.

Количество ветвей в программе обычно пропорционально ее размеру. После интеграции программных модулей в систему, методы структурного тестирования оказываются невыполнимыми. Поэтому методы тестирования ветвей, как правило, используются при тестировании отдельных программных элементов и модулей.

При тестировании ветвей не проверяются все возможные комбинации ветвей программы. Не считая самых тривиальных программных компонентов без циклов, подобная полиая проверка компонента оказывается нереальной, так как в программах с циклами существует бесконечное число возможных комбинаций ветвей. В программе могут быть дефекты, которые проявляются только при определенных комбинациях ветвей, даже если все операторы программы протестированы (т.е. выполнились) хотя бы один раз.

Метод тестирования ветвей основывается на графе потоков управления программы. Этот граф представляет собой скелетную модель всех ветвей программы. Граф потоков управления состоит из узлов, соответствующих ветвлениям решений, и дуг, показывающих поток управления. Если в программе нет операторов безусловного перехода, то создание графа – достаточно простой процесс. При построении графа потоков все последовательные операторы (операторы присвоения, вызова процедур и ввода-вывода) можно проигнорировать. Каждое ветвление операторов условного перехода (if-then-else или case) представлено отдельной ветвью, а циклы обозначаются стрелками, концы которых замкнуты на узле с условием цикла. На рис. 20.8 показаны циклы и ветвления в графе потоков управления программы бинарного поиска.

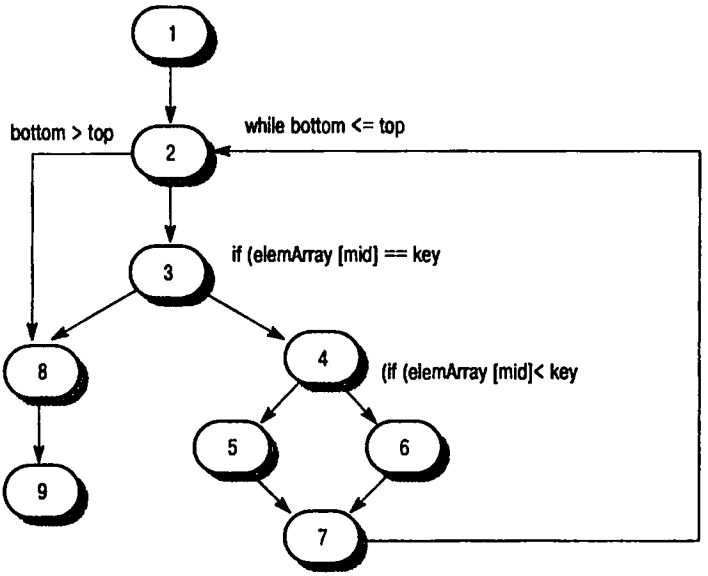


Рис. 20.8. Граф потоков управления программы бинарного поиска

Цель структурного тестирования – удостовериться, что каждая независимая ветвь программы выполняется хотя бы один раз. Независимая ветвь программы – это ветвь, которая проходит по крайней мере по одной новой дуге графа потоков. В терминах программы это означает ее выполнение при новых условиях. С помощью трассировки в графе потоков управления программы бинарного поиска можно выделить следующие независимых ветвей.

1, 2, 3, 8, 9
1, 2, 3, 4, 6, 7, 2
1, 2, 3, 4, 5, 7, 2
1, 2, 3, 4, 6, 7, 2, 8, 9

Если все эти ветви выполняются, можно быть уверенным в том, что, во-первых, каждый оператор выполняется по крайней мере один раз и, во-вторых, каждая ветвь выполняется при условиях, принимающих как истинные, так и ложные значения.

Количество независимых ветвей в программе можно определнть, вычислив цикломатическое число графа потоков управления программы [232]. Цикломатическое число C любого связанного графа G вычисляется по формуле

C (G) = количество дуг – количество узлов + 2.

Для программ, не содержащих операторов безусловного перехода, значение цикломатического числа всегда больше количества проверяемых условий. В составных условиях, содержащих более одного логического оператора, следует учитывать каждый логический оператор. Например, если в программе шесть операторов if и один цикл while, то цикломатическое число равно 8. Если одно условное выражение является составным выражением с двумя логическими операторами (объединенными операторами and или or), то цикломатическое число будет равно 10. Цикломатическое число программы бинарного поиска равно 4.

После определения количества независимых ветвей в программе путем вычисления цикломатического числа разрабатываются контрольные тесты для проверки каждой ветви. Минимальное количество тестов, требующееся для проверки всех ветвей программы, равно цикломатическому числу.

Проектирование контрольных тестов для программы бинарного поиска не вызывает затруднений. Однако, если программы имеют сложную структуру ветвлений, трудно предсказать, как будет выполняться какой-либо отдельный контрольный тест. В таких случаях используется динамический анализатор программ для составления рабочего профиля программы.

Динамические анализаторы программ — это инструментальные средства, которые работают совместно с компиляторами. Во время компилирования в сгенерированный код добавляются дополнительные инструкции, подсчитывающие, сколько раз выполняется каждый оператор программы. Чтобы при выполнении отдельных контрольных тестов увидеть, какие ветви в программе выполнялись, а какие нет, распечатывается рабочий профиль программы, где видны непроверенные участки.

20.2. Тестирование сборки

После того как протестированы все отдельные программные компоненты, выполняется сборка системы, в результате чего создается частичная или полная система. Процесс интеграции системы включает сборку и тестирования полученной системы, в ходе которого выявляются проблемы, возникающие при взаимодействии компонентов. Тесты, проверяющие сборку системы, должны разрабатываться на основе системной спецификации, причем тестирование сборки следует начинать сразу после создания работоспособных версий компонентов системы.

Во время тестирования сборки возникает проблема локализации выявленных ошибок. Между компонентами системы существуют сложные взаимоотношения, и при обнаружении аномальных выходных данных бывает трудно установить источник ошибки. Чтобы облегчить локализацию ошибок, следует использовать пошаговый метод сборки и тести-

рованиия системы. Сначала следует создать минимальную конфигурацию системы и ее протестировать. Затем в минимальную конфигурацию нужно добавить новые компоненты и снова протестировать, и так далее до полной сборки системы.

В примере на рис. 20.9 последовательность тестов T1, T2 и T3 сначала выполняется в системе, состоящей из модулей A и B (минимальная конфигурация системы). Если во время тестирования обнаружены дефекты, они исправляются. Затем в систему добавляется модуль C. Тесты T1, T2 и T3 повторяются, чтобы убедиться, что в новой системе нет никаких неожиданных взаимодействий между модулями A и B. Если в ходе тестирования появились какие-то проблемы, то, вероятно, они возникли во взаимодействиях с новым модулем C. Источник проблемы локализован, таким образом упрощается определение дефекта и его исправление. Затем система запускается с тестами T4. На последнем шаге добавляется модуль D и система тестируется еще раз выполняемыми ранее тестами, а затем новыми тестами T5.

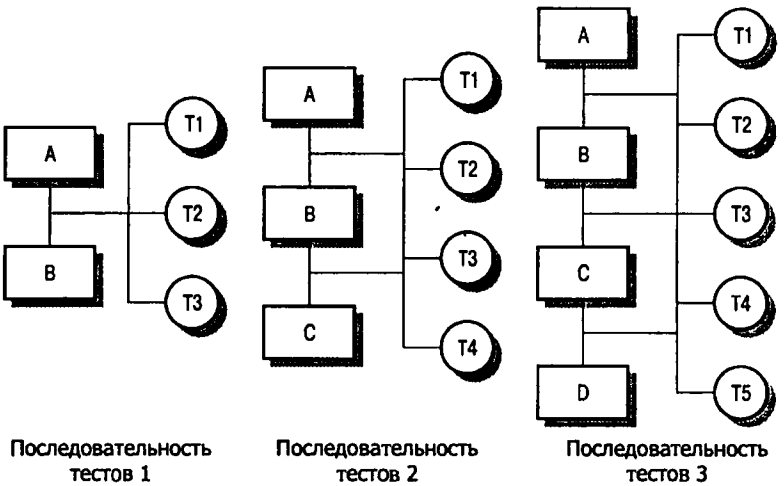


Рис. 20.9. Тестирование сборки

Конечно, на практике редко встречаются такие простые модели. Функции системы могут быть реализованы в нескольких компонентах. Тестирование новой функции, таким образом, требует интеграции сразу нескольких компонентов. В этом случае тестирование может выявить ошибки во взаимодействиях между этими компонентами и другими частями системы. Исправление ошибок может оказаться сложным, так как в данном случае ошибки влияют на целую группу компонентов, реализующих конкретную функцию. Более того, при интеграции нового компонента может измениться структура взаимосвязей между уже протестированными компонентами. Вследствие этого могут выявиться ошибки, которые не были выявлены при тестировании более простой конфигурации.

20.2.1. Нисходящее и восходящее тестирование

Методики нисходящего и восходящего тестирования (рис. 20.10) отражают разные подходы к системной интеграции. При нисходящей интеграции компоненты высокого уровня интегрируются и тестируются еще до окончания их проектирования и реализации. При восходящей интеграции перед разработкой компонентов более высокого уровня сначала интегрируются и тестируются компоненты нижнего уровня.

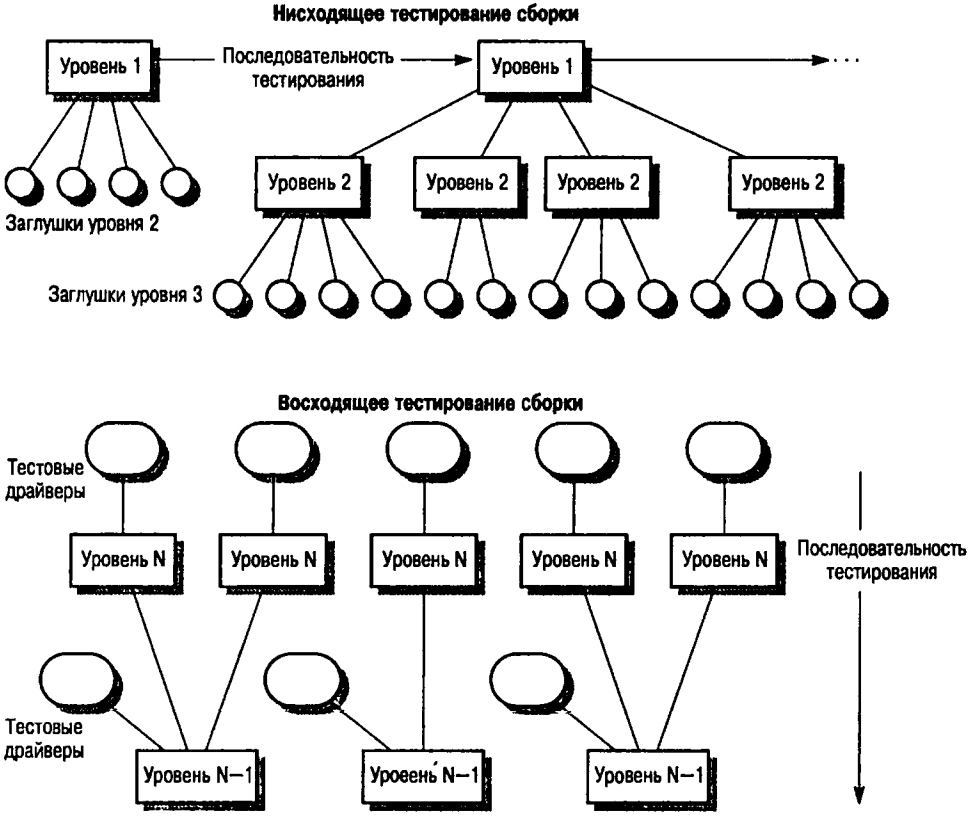


Рис. 20.10. Нисходящее и восходящее тестирование сборки

Нисходящее тестирование является неотъемлемой частью процесса нисходящей разработки систем, при котором сначала разрабатываются компоненты верхнего уровня, а затем компоненты, находящиеся на нижних уровнях иерархии. Программу можно представить в виде одного абстрактного компонента с субкомпонентами, являющимися заглушками. Заглушки имеют такой же интерфейс, что и компонент, но с ограниченной функциональностью. После того как компонент верхнего уровня запрограммирован и протестирован, таким же образом реализуются и тестируются его субкомпоненты. Процесс продолжается до тех пор, пока не будут реализованы компоненты самого нижнего уровня. Затем вся система тестируется целиком.

При восходящем тестировании, наоборот, сначала интегрируются и тестируются модули, расположенные на более низких уровнях иерархии. Затем выполняется сборка и тестирование модулей, расположенных на верхнем уровне иерархии, и так до тех пор, пока не будет протестирован последний модуль. При таком подходе не требуется наличие законченного архитектурного проекта системы, и поэтому он может начинаться на раннем этапе процесса разработки. Обычно такой подход применяется тогда, когда в системе есть повторно используемые компоненты или модифицированные компоненты из других систем.

Нисходящее и восходящее тестирование можно сравнить по четырем направлениям.

1. *Верификация и аттестация системной архитектуры.* При нисходящем тестировании больше возможностей выявить ошибки в архитектуре системы на раннем этапе

процесса разработки. Обычно это структурные ошибки, раннее выявление которых предполагает их исправление без дополнительных затрат. При восходящем тестировании структура высокого уровня не утверждается вплоть до последнего этапа разработки системы.

2. *Демонстрация системы.* При нисходящей разработке незаконченная система вполне пригодна для работы уже на ранних этапах разработки. Этот факт является важным психологическим стимулом использования нисходящей модели разработки систем, поскольку демонстрирует осуществимость управления системой. Аттестация проводится в начале процесса тестирования путем создания демонстрационной версии системы. Но если система создается из повторно используемых компонентов, то и при восходящей разработке также можно создать ее демонстрационную версию.
3. *Реализация тестов.* Нисходящее тестирование сложно реализовать, так как необходимо моделировать программы-заглушки нижних уровней. Программы-заглушки могут быть упрощенными версиями представляемых компонентов. При восходящем тестировании для того, чтобы использовать компоненты нижних уровней, необходимо разработать тестовые драйверы, которые эмулируют окружение компонента в процессе тестирования.
4. *Наблюдение за ходом испытаний.* При нисходящем и восходящем тестировании могут возникать проблемы, связанные с наблюдениями за ходом тестирования. В большинстве систем, разрабатываемых сверху вниз, более верхние уровни системы, которые реализованы первыми, не генерируют выходные данные, однако для проверки этих уровней нужны какие-либо выходные результаты. Испытатель должен создать искусственную среду для генерации результатов теста. При восходящем тестировании также может возникнуть необходимость в создании искусственной среды (тестовых драйверов) для исследования компонентов нижних уровней.

На практике при разработке и тестировании систем чаще всего используется композиция восходящих и нисходящих методов. Разные сроки разработки для разных частей системы предполагают, что группа, проводящая тестирование и интеграцию, должна работать с каким-либо готовым компонентом. Поэтому во время процесса тестирования сборки в любом случае необходимо разрабатывать как заглушки, так и тестовые драйверы.

20.2.2. Тестирование интерфейсов

Как правило, тестирование интерфейса выполняется в тех случаях, когда модули или подсистемы интегрируются в большие системы. Каждый модуль или подсистема имеет заданный интерфейс, который вызывается другими компонентами системы. Цель тестирования интерфейса — выявить ошибки, возникающие в системе вследствие ошибок в интерфейсах или неправильных предположений об интерфейсах.

Схема тестирования интерфейса показана на рис. 20.11. Стрелки в верхней части схемы означают, что контрольные тесты применяются не к отдельным компонентам, а к подсистемам, полученным в результате комбинирования этих компонентов.

Данный тип тестирования особенно важен в объектно-ориентированном проектировании, в частности при повторном использовании объектов и классов объектов. Объекты в значительной степени определяются с помощью интерфейсов и могут повторно использоваться в различных комбинациях с разными объектами и в разных системах. Во время тестирования отдельных объектов невозможно выявить ошибки интерфейса, так как они являются скорее результатом взаимодействия между объектами, чем изолированного поведения одного объекта.

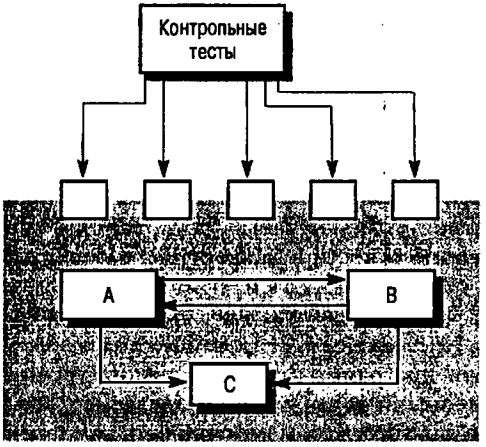


Рис. 20.11. Тестирование интерфейсов

Между компонентами программы могут быть разные типы интерфейсов и соответственно разные типы ошибок интерфейсов.

1. *Параметрические интерфейсы.* Интерфейсы, в которых ссылки на данные и иногда функции передаются в виде параметров от одного компонента к другому.
2. *Интерфейсы разделяемой памяти.* Интерфейсы, в которых какой-либо блок памяти совместно используется разными подсистемами. Одна подсистема помещает данные в память, а другие подсистемы используют эти данные.
3. *Процедурные интерфейсы.* Интерфейсы, в которых одна подсистема инкапсулирует набор процедур, вызываемых из других подсистем. Такой тип интерфейса имеют объекты и абстрактные типы данных.
4. *Интерфейсы передачи сообщений.* Интерфейсы, в которых одна подсистема запрашивает сервис у другой подсистемы посредством передачи ей сообщения. Ответное сообщение содержит результаты выполнения сервиса. Некоторые объектно-ориентированные системы имеют такой тип интерфейсов; например, так работают системы клиент/сервер.

Ошибки в интерфейсах являются наиболее распространенными типами ошибок в сложных системах [226] и делятся на три класса.

- *Неправильное использование интерфейсов.* Компонент вызывает другой компонент и совершает ошибку при использовании его интерфейса. Данный тип ошибки особенно распространен в параметрических интерфейсах; например, параметры могут иметь неправильный тип, следовать в неправильном порядке или же иметь неверное количество параметров.
- *Неправильное понимание интерфейсов.* Вызывающий компонент, в который заложена неправильная интерпретация спецификации интерфейса вызываемого компонента, предполагает определенное поведение этого компонента. Если поведение вызываемого компонента не совпадает с ожидаемым, поведение вызывающего компонента становится непредсказуемым. Например, если программа бинарного поиска вызывается для поиска заданного элемента в неупорядоченном массиве, то в работе программы произойдет сбой.

- *Ошибки синхронизации.* Такие ошибки встречаются в системах реального времени, где используются интерфейсы разделяемой памяти или передачи сообщений. Подсистема — производитель данных и подсистема — потребитель данных могут работать с разной скоростью. Если при проектировании интерфейса не учитывать этот фактор, потребитель может, например, получить доступ к устаревшим данным, потому что производитель к тому моменту еще не успел обновить совместно используемые данные.

Тестирование дефектов интерфейсов сложно, поскольку некоторые ошибки могут проявиться только в необычных условиях. Например, пусть некий объект реализует очередь в виде структуры списка фиксированного размера. Вызывающий его объект при вводе очередного элемента не проверяет переполнение очереди, так как предполагает, что очередь реализована как структура неограниченного размера. Такую ситуацию можно обнаружить только во время выполнения специальных тестов: специально вызывается переполнение очереди, которое приводит к непредсказуемому поведению объекта.

Другая проблема может возникнуть из-за взаимодействий между ошибками в разных программных модулях или объектах. Ошибки в одном объекте можно выявить только тогда, когда поведение другого объекта становится непредсказуемым. Например, для получения сервиса один объект вызывает другой объект и полагает, что полученный ответ правильный. Если объект неправильно понимает вычисленные значения, возвращаемое значение может быть достоверным, но неправильным. Такие ошибки можно выявить только тогда, когда оказываются неправильными дальнейшие вычисления.

Вот несколько общих правил тестирования интерфейсов.

1. Просмотрите тестируемый код и составьте список всех вызовов, направленных к внешним компонентам. Разработайте такие наборы тестовых данных, при которых параметры, передаваемые внешним компонентам, принимают крайние значения из диапазонов их допустимых значений. Использование экстремальных значений параметров с высокой вероятностью обнаруживает несоответствия в интерфейсах.
2. Если между интерфейсами передаются указатели, всегда тестируйте интерфейс с нулевыми параметрами указателя.
3. При вызове компонента через процедурный интерфейс используйте тесты, вызывающие сбой в работе компонента. Одна из наиболее распространенных причин ошибок в интерфейсе — неправильное понимание спецификации компонентов.
4. В системах передачи сообщений используйте тесты с нагрузкой, которые рассматриваются в следующем разделе. Разрабатывайте тесты, генерирующие в несколько раз большее количество сообщений, чем будет в обычной работе системы. Эти же тесты позволяют обнаружить проблемы синхронизации.
5. При взаимодействии нескольких компонентов через разделяемую память разрабатывайте тесты, которые изменяют порядок активизации компонентов. С помощью таких тестов можно выявить сделанные программистом неявные предположения о порядке использования компонентами разделяемых данных.

Обычно статические методы тестирования более рентабельны, чем специальное тестирование интерфейсов. В языках со строгим контролем типов, например Java, многие ошибки интерфейсов помогает обнаружить компилятор. В языках со слабым контролем типов (например, C) ошибки интерфейса может выявить статический анализатор, такой как LINT (см. главу 19). Кроме того, при инспектировании программ можно сосредоточиться именно на проверке интерфейсов компонентов.

20.2.3. Тестирование с нагрузкой

После полной интеграции системы можно оценить такие интеграционные свойства системы (см. главу 2), как производительность и надежность. Чтобы убедиться, что система может работать с заданной нагрузкой, разрабатываются тесты для измерения производительности. Обычно планируются серии тестов с постоянным увеличением нагрузки, пока производительность системы не начнет снижаться.

Некоторые классы систем проектируются с учетом работы под определенной нагрузкой. Например, система обработки транзакций проектируется так, чтобы обрабатывать 100 транзакций в секунду; сетевая операционная система — чтобы обрабатывать информацию от 200 отдельных терминалов. При тестировании с нагрузкой выполнение тестов начинается с максимальной нагрузки, указанной в проекте системы, и продолжается до тех пор, пока не произойдет сбой в работе системы. Данный тип тестирования выполняет две функции.

1. Тестируется поведение системы во время сбоя. В процессе эксплуатации могут возникать ситуации, при которых нагрузка в системе превышает максимально допустимую. В таких ситуациях очень важно, чтобы сбой в системе не приводил к нарушению целостности данных или к потере сервисных возможностей.
2. Чтобы выявить дефекты, которые не проявляются в обычных режимах работы, система подвергается тестированию с нагрузкой. Хотя подобные дефекты не приводят к ошибкам при обычном использовании системы, на практике могут возникнуть необычные комбинации стандартных условий; именно они воспроизводятся во время тестирования с нагрузкой.

Тестирование с нагрузкой чаще всего применяется в распределенных системах. В таких системах при большой нагрузке сеть порой "забивается" данными, которыми обмениваются разные процессы. Постепенно процессы все больше замедляются, поскольку они ожидают данные запросов от других процессов.

20.3. Тестирование объектно-ориентированных систем

Мы рассмотрели два основных подхода к тестированию программного обеспечения — компонентное тестирование, при котором компоненты системы тестируются независимо друг от друга, и тестирование сборки, когда компоненты интегрированы в подсистемы и тестируется конечная система. Эти подходы в равной мере применимы и к объектно-ориентированным системам. Однако системы, разработанные по функциональной модели, и объектно-ориентированные системы имеют существенные отличия.

1. Объекты, как отдельные программные компоненты, представляют собой нечто большее, чем отдельные подпрограммы или функции.
2. Объекты, интегрированные в подсистемы, обычно слабо связаны между собой и поэтому сложно определить "самый верхний уровень" системы.
3. При анализе повторно используемых объектов их исходный код может быть недоступным для испытателей.

Эти отличия означают, что при проверке объектов можно применять тестирование методом белого ящика, основанное на анализе кода, а при тестировании сборки следует

использовать другие подходы. Применительно к объектно-ориентированным системам можно определить четыре уровня тестирования.

1. *Тестирование отдельных методов (операций), ассоциированных с объектами.* Обычно методы представляют собой функции или процедуры. Поэтому здесь можно использовать тестирование методами черного и белого ящиков, которые рассматривались ранее.
2. *Тестирование отдельных классов объектов.* Принцип тестирования методом черного ящика остается без изменений, однако, понятие "класса эквивалентности" необходимо расширить. Тестирование классов объектов обсуждается в разделе 20.3.1.
3. *Тестирование кластеров объектов.* Нисходящая и восходящая сборки оказываются не пригодными для создания групп связанных объектов. Поэтому здесь следует применять другие методы тестирования, например основанные на сценариях. Эти методы рассматриваются в разделе 20.3.2.
4. *Тестирование системы.* Верифнкация и аттестация объектно-ориентированной системы выполняется точно так же, как и для любых других типов систем.

В настоящее время методы тестирования объектно-ориентированных систем достаточно хорошо разработаны [39, 20*]. В следующих разделах приведен обзор основных методов тестирования объектно-ориентированных систем.

20.3.1. Тестирование классов объектов

Подход к тестовому покрытию систем, описанный в разделе 20.1.3, требует, чтобы все операторы в программе выполнялись хотя бы один раз, а также чтобы выполнялись все ветви программы. При тестировании объектов полное тестовое покрытие включает:

- раздельное тестирование всех методов, ассоциированных с объектом;
- проверку всех атрибутов, ассоциированных с объектом;
- проверку всех возможных состояний объекта (для этого необходимо моделирование событий, приводящих к изменению состояния объекта).

В качестве примера возьмем метеорологическую станцию, которая рассматривалась в главе 12. Интерфейс объекта, соответствующего метеостанции, показан на рис. 20.12. У этого объекта только один атрибут, который является его идентификатором. Это константа, значение которой необходимо задать после инсталляции объекта. Таким образом, нужен тест, который проверял бы задание идентификатора. Необходимо также определить контрольные тесты для методов **отчет**, **калибровать**, **тестировать**, **запуск** и **завершение**. В идеале следует протестировать все эти методы независимо, но в некоторых случаях нужна последовательность тестов. Например, чтобы протестировать метод **завершение**, необходимо сначала выполнить метод **запуск**.

При тестировании состояний объекта **Метеостанция** используется модель состояний, показанная на рис. 12.12. С помощью этой модели можно определить последовательность состояний, которые нужно протестировать. В принципе следует проверить каждый возможный переход из одного состояния в другое, хотя на практике такой подход оказывается слишком дорогостоящим. В нашем примере необходимо протестировать такие последовательности состояний:

Останов → Ожидание → Останов
Ожидание → Калибровка → Тестирование → Передача → Ожидание
Ожидание → Калибровка → Ожидание → Обобщение → Передача → Ожидание

Использование наследования усложняет разработку тестов для классов объектов. Если класс предоставляет методы, унаследованные от подклассов, то необходимо протестировать все подклассы со всеми унаследованными методами. Понятие классов эквивалентности можно применить также и к классам объектов. Здесь тестовые данные из одного класса эквивалентности тестируют одни и те же свойства объектов.

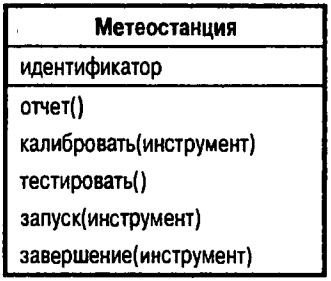


Рис. 20.12. Интерфейс объекта метеорологической станции

20.3.2. Интеграция объектов

При разработке объектно-ориентированных систем различия между уровнями интеграции менее заметны, поскольку методы и данные компонуются (интегрируются) в виде объектов и классов объектов. Тестирование классов объектов соответствует тестированию отдельных элементов. В объектно-ориентированных системах нет непосредственного эквивалента тестированию модулей. Однако считается, что группы классов, которые совместно предоставляют набор сервисов, следует тестировать вместе [245]. Такой вид тестирования называется *тестированием кластеров*.

Для объектно-ориентированных систем не подходит ни восходящая, ни нисходящая интеграция системы, поскольку здесь нет строгой иерархии объектов. Поэтому создание кластеров основывается на выделении методов и сервисов, реализуемых посредством этих кластеров. При тестировании сборки объектно-ориентированных систем используется три подхода.

1. *Тестирование сценариев и вариантов использования.* Варианты использования или сценарии (см. главу 6) описывают какой-либо один режим работы системы. Тестирование может базироваться на описании этих сценариев и кластеров объектов, реализующих данный вариант использования.
2. *Тестирование потоков.* Этот подход основывается на проверке системных откликов на ввод данных или группу входных событий. Объектно-ориентированные системы, как правило, событийно-управляемые, поэтому для них особенно подходит данный вид тестирования. При использовании этого подхода необходимо знать, как в системе проходит обработка потоков событий.
3. *Тестирование взаимодействий между объектами.* Это метод тестирования групп взаимодействующих объектов, предложенный в работе [194]. Этот промежуточный уровень тестирования сборки системы основан на определении путей "метод–сообщение", отслеживающих последовательности взаимодействий между объектами.

Тестирование сценариев часто оказывается более эффективным, чем другие методы тестирования. Сам процесс тестирования можно спланировать так, чтобы в первую очередь проверялись наиболее вероятные сценарии и только затем исключительные сцена-

рии. Поэтому тестирование сценариев удовлетворяет основному принципу, согласно которому при тестировании больше внимания необходимо уделять наиболее часто используемым частям системы.

Чтобы проиллюстрировать тестирование сценариев, вновь рассмотрим систему метеостанции. Сценарии можно определить, исходя из разработанных вариантов использования, однако их может быть недостаточно для тестирования. Поэтому при планировании тестирования можно использовать диаграммы взаимодействия, отображающие реализацию вариантов использования посредством системных объектов.

Рассмотрим рис. 20.13 (взятый из главы 12), на котором изображена последовательность операций, выполняемых при сборе метеоданных. Эту диаграмму можно использовать для определения тестируемых операций и для разработки тестовых сценариев.

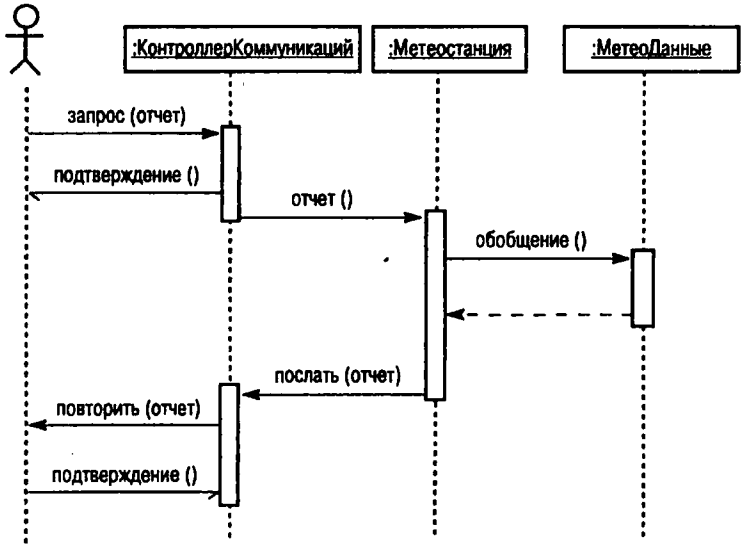


Рис. 20.13. Диаграмма последовательности сбора метеоданных

На рис. 20.13 видно, что при получении запроса на отчет о метеоданных в системе будет выполняться следующий поток методов:

КонтроллерКоммуникаций:запрос → Метеостанция:отчет → МетеоДанные:обобщение

После выбора сценариев для тестирования системы важно убедиться, что все методы каждого класса будут выполняться хотя бы один раз. Для этого можно составить технологическую карту проверок классов объектов и методов и при выборе сценария отмечать выполняемый метод. Конечно, все комбинации методов выполнить невозможно, но по крайней мере можно удостовериться, что все методы протестированы как часть какой-либо последовательности выполняемых методов.

20.4. Инструментальные средства тестирования

Тестирование — дорогой и трудоемкий этап разработки программных систем. Поэтому создан широкий спектр инструментальных средств для поддержки процесса тестирования, которые значительно сокращают расходы на него.

На рис. 20.14 показаны возможные инструментальные средства тестирования и отношения между ними. Перечислим их.

1. *Организатор тестов.* Управляет выполнением тестов. Он отслеживает тестовые данные, ожидаемые результаты и тестируемые функции программы.
2. *Генератор тестовых данных.* Генерирует тестовые данные для тестируемой программы. Он может выбирать тестовые данные из базы данных или использовать специальные шаблоны для генерации случайных данных необходимого вида.
3. *Оракул.* Генерирует ожидаемые результаты тестов. В качестве оракулов могут выступать предыдущие версии программы или исследуемого объекта. При тестировании параллельно запускаются оракул и тестируемая программа и сравниваются результаты их выполнения.
4. *Компаратор файлов.* Сравнивает результаты тестирования с результатами предыдущего тестирования и составляет отчет об обнаруженных различиях. Компараторы особенно важны при сравнении различных версий программы. Различия в результатах указывают на возможные проблемы, существующие в новой версии системы.
5. *Генератор отчетов.* Формирует отчеты по результатам проведения тестов.
6. *Динамический анализатор.* Добавляет в программу код, который подсчитывает, сколько раз выполняется каждый оператор. После запуска теста создает исполняемый профиль, в котором показано, сколько раз в программе выполняется каждый оператор.
7. *Имитатор.* Существует несколько типов имитаторов. Целевые имитаторы моделируют машину, на которой будет выполняться программа. Имитатор пользовательского интерфейса — это программа, управляемая сценариями, которая моделирует взаимодействия с интерфейсом пользователя. Имитатор ввода-вывода генерирует последовательности повторяющихся транзакций.

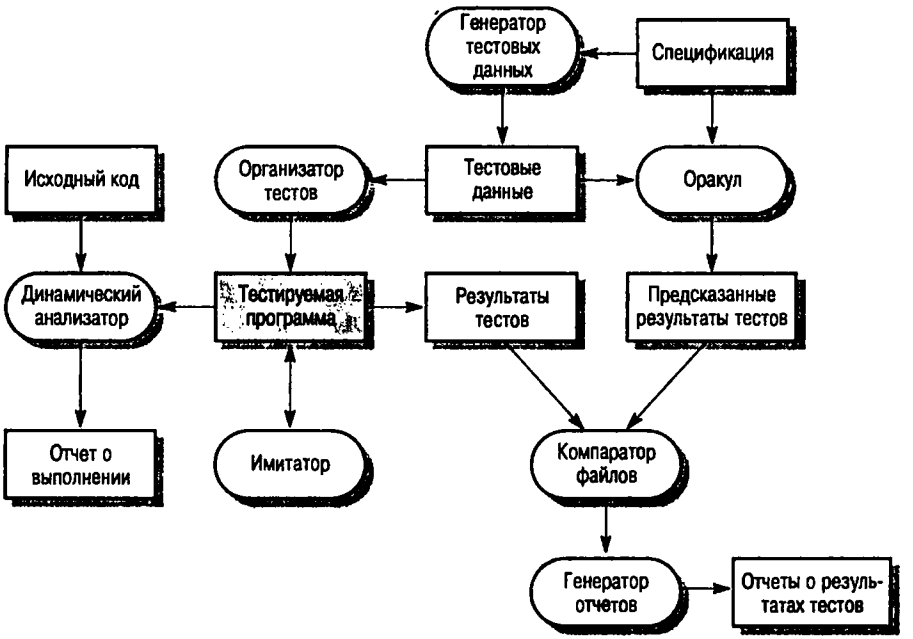


Рис. 20.14. Инструментальные средства тестирования

Требования, предъявляемые к процессу тестирования больших систем, зависят от типа разрабатываемого приложения. Поэтому инструментальные средства тестирования неизменно приходится адаптировать к процессу тестирования конкретной системы.

Для создания полного комплекса инструментальиого средства тестирования, как правило, требуется много сил и времени. Весь набор инструментальных средств, показанных на рис. 20.14, используется только при тестировании больших систем. Для таких систем полная стоимость тестирования может достигать 50% от всей стоимости разработки системы. Вот почему выгодно инвестировать разработку высококачественных и производительных CASE-средств тестирования.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- При тестировании систем прежде всего тестируются наиболее часто используемые части системы, а затем части, используемые реже.
- Области эквивалентности входных данных являются одним из способов представления сценариев тестирования. Часто значения, которые с наибольшей вероятностью ведут к результативному тестированию, лежат на границе областей эквивалентности.
- При тестировании методом черного ящика не требуется доступ к исходному коду. Тестовые данные получают на основе программной спецификации.
- Во время тестирования системной сборки исследуются взаимодействия между компонентами системы и интерфейсами компонентов.
- Дефекты интерфейса часто возникают из-за ошибок, сделанных при интерпретации спецификации требований. Тестирование интерфейсов должно обнаруживать дефекты в интерфейсах объектов и модулей.
- При тестировании классов объектов следует так проектировать тесты, чтобы проверить все методы, ассоциированные с классами, определить и оценить все свойства объектов и протестировать объект во всех возможных состояниях.

Упражнения

20.1. Обсудите различия между тестированием методом черного ящика и структурным тестированием. Подумайте, каким образом можно совместно использовать эти методы в процессе тестирования дефектов.

20.2. Какие проблемы тестирования могут возникнуть в программах, которые обрабатывают как очень большие, так и очень малые числа?

20.3. Разработайте набор тестовых данных для следующих компонентов:

- программа сортировки массивов целых чисел;
- программа, которая вычисляет количестао символов (отличных от пробелов) в текстовых строках;
- программа, которая проверяет текстовые строки и заменяет последовательности пробелов одним пробелом;
- объект, реализующий символьные строки разной длины. Среди операций, ассоциированных с этим объектом, должны быть операция конкатенации, операция определения длины строки и операция выбора подстроки.

20.4. Напишите код для первых трех перечисленных выше программ, используя любой язык программирования. Для каждой программы рассчитайте цикломатическое число.

20.5. На примере небольшой программы покажите, почему практически невозможно полностью протестировать программу.

20.6. Для проверки созданных вами программ, разработайте контрольные тесты в дополнение к тем, которые уже были рассмотрены. Выявил ли анализ кода упущения в первоначальных наборах тестовых данных?

20.7. Реализуйте (на языке Java или C++) класс объектов SYMBOL_TABLE, который будет использоваться как часть системы компиляции. Этот класс должен иметь следующие методы: добавление имени и типа данных в таблицу идентификаторов, удаление имени, изменение информации, связанной с именем, и поиск по таблице. Организуйте инспектирование кода этого объекта (см. главу 19) и сделайте подсчет обнаруженных ошибок. Протестируйте объект методом черного ящика и сравните ошибки, выявленные при тестировании и при инспектировании.

20.8. Объясните, почему методы нисходящего и восходящего тестирования не подходят для объектно-ориентированных систем.

20.9. Создайте сценарии тестирования состояний микроволновой печи, модель состояний которой показана на рис. 7.5.

21 Аттестация критических систем

Цели

В настоящей главе рассматриваются методы верификации и аттестации критических систем. Прочитав эту главу, вы должны:

- знать преимущества и недостатки использования формальных методов верификации критических систем;
- иметь представление о методах оценивания безотказности программных системы и модели возрастания безотказности;
- познакомиться с основной идеей доказательства безопасности и уметь доказывать невозможность появления аварийных ситуаций в системе.

Содержание

Верификация и аттестация критической системы имеют много общего с подобными процессами, выполняемыми над любой другой программной системой. В ходе этих процессов проверяется соответствие системы своей спецификации и соответствие поведения системы требованиям заказчика. Однако природа критических систем такова, что, как правило, в дополнение к обычному анализу и тестированию системы необходимы еще процессы доказательства ее надежности. Это требуется по двум причинам.

1. *Цена отказа критических систем.* В критических системах стоимость отказа и его последствия потенциально значительно выше, чем в других системах. Поэтому экономически выгоднее выделить достаточно средств на верификацию и аттестацию критических систем, чтобы обнаружить и исправить возможные ошибки еще в процессе разработки системы.
2. *Аттестация свойств функциональной надежности.* Заказчики критических систем должны быть уверены в том, что система соответствует определенным показателям функциональной надежности (работоспособность, безотказность, безопасность и защищенность). Для оценки этих свойств требуется специальный процесс верификации и аттестации, который рассматривается далее в главе.

По указанным выше причинам стоимость процесса верификации и аттестации для критических систем обычно значительно выше, чем для других систем. Нет ничего удивительного в том, что на верификацию и аттестацию критических систем приходится более 50% от полной стоимости разработки. Конечно, такие расходы оправдывают себя, если удается избежать дорогостоящих отказов системы. Например, в 1996 году вследствие отказа программного обеспечения, установленного на ракете Ariane 5, потеряно несколько спутников, стоимостью сотни миллионов долларов.

В процессе аттестации критических систем кроме непосредственной аттестации систем должна также проводиться аттестация процессов, используемых при ее разработке. Как отмечается в главах 8 и 24, на качество системы влияет качество процессов разработки ПО. Поэтому при создании систем с высокими требованиями надежности необходимо прежде всего обеспечить четкое соблюдение стандартов процесса разработки.

Такая аттестация процесса разработки является неотъемлемой частью стандартов управления качеством ISO 9000[1], кратко рассмотренных в главе 24. Согласно этим стандартам, на все используемые процессы разработки и связанные с ними процессы, обеспечивающие их соблюдение, должна быть документация. Как правило, требуется иметь систему сертификации полноты и завершенности процессов разработки и проверки качества промежуточных и конечного продуктов, например в виде специальных форм. Стандарты ISO 9000 определяют, какие промежуточные продукты процесса должны производиться и контролироваться и кто отвечает за их производство и проверку. Использование форм, которые описывают процесс анализа системных отказов, рассматривается в разделе 21.3.3.

21.1. Формальные методы и критические системы

В организациях, занимающихся разработкой критических систем, все еще продолжаются дебаты о роли формальных методов в процессе разработки систем с критическими требова-

[1] *Международная организация по стандартизации (International Standarts Organization – ISO), являясь ассоциацией национальных организаций по стандартизации, обеспечивает разработку и поддержку международных стандартов в сфере коммуникаций и обмена информацией.* – Прим. ред.

ниями к безопасности и надежности. Использование формальной математической спецификации и связанной с ней верификации является стандартом в Великобритании для систем, критических по обеспечению безопасности [241]. Однако большинство разработчиков критических систем считают формальные методы неэффективными и уверены, что их использование может привести даже к снижению (а не повышению) надежности системы.

Формальные методы применяются на двух уровнях разработки критических систем.

1. Разработка формальной спецификации системы и математический анализ противоречий в формальной спецификации. Как отмечалось в главе 9, данная методика эффективно выявляет ошибки и упущения в системной спецификации. В этой главе использование формального подхода рассматривается на примере спецификации части системы, отвечающей за инъекции инсулина, которая описана в главе 16.
2. Формальная верификация для проверки соответствия программного кода системной спецификации. Для формальной верификации нужна формальная спецификация. Формальная верификация эффективно выявляет ошибки, сделанные на этапах программирования и проектирования системы.

Аргумент за использование формальной спецификации и связанной с ней верификации программ состоит в том, что во время формального специфицирования выполняется тщательный и подробный анализ требований. Этот анализ позволяет выявить потенциальные противоречия и упущения, которые в противном случае могут оказаться незамеченными вплоть до появления рабочей версии системы. Формальная верификация демонстрирует соответствие разрабатываемой программы спецификации.

Против использования формальной спецификации выдвигается следующий аргумент: ее использование требует специальной системы нотаций, пользоваться которой может только специально обученный персонал. Используемая для формальной спецификации система нотаций может оказаться непонятной специалистам в той предметной области, где будет эксплуатироваться система. Поэтому проблемы согласования требований необходимо решить с помощью специальных формальных методов. В этой ситуации разработчики ПО не могут распознать потенциальные проблемы требований, поскольку не являются специалистами в предметной области приложения; специалисты в области приложения также не могут обнаружить проблемы, так как они не понимают спецификацию. Поэтому, несмотря на то что спецификация может быть математически корректной, свойства системы, которые требуются а действительности, в спецификации могут быть не определены.

Верификация нетривиальной системы ПО занимает много времени и требует применения специальных средств и методов, например методов доказательства теорем и математического оценивания. Поэтому верификация является чрезвычайно дорогостоящим процессом, причем с увеличением размеров системы расходы на формальную верификацию возрастают нелинейно. Именно поэтому многие считают формальную верификацию экономически не выгодной. Такого же уровня надежности или безопасности можно достигнуть с меньшими затратами, используя такие методы верификации, как инспектирование и тестирование системы. Такое утверждение пока невозможно ни подтвердить, ни опровергнуть, поэтому при разработке некоторых систем все равно применяются формальные методы.

Иногда говорят, что использование формальных методов в процессе разработки системы позволяет создать более надежные и безотказные системы. Несомненно, формальная спецификация содержит меньше противоречий, которые должен разрешать разра-

ботчик. Однако формальная спецификация и доказательство не гарантируют, что система окажется надежной на практике, чему есть несколько причин.

1. *Спецификация может не отражать реальных требований пользователей системы.* Установлено, что многие ошибки являются следствием ошибок и упущений в системной спецификации, не обнаруженных при создании формальной спецификации [226]. Кроме того, пользователям часто непонятны формальные нотации, поэтому они не могут непосредственно проанализировать формальную спецификацию и обнаружить в ней ошибки и упущения.
2. *В доказательствах могут быть ошибки.* Доказательства больших и сложных программ, как правило, также большие и сложные, поэтому часто содержат ошибки.
3. *При доказательстве применяется неверный шаблон использования.* Если система используется неверно, то доказательство также может быть неверным.

Несмотря на все эти недостатки, я полагаю, что формальные методы играют важную роль в разработке надежного и безопасного ПО. Формальные спецификации весьма эффективно выявляют проблемы требований, которые являются наиболее распространенными ошибками. Формальная верификация повышает уровень надежности наиболее критических компонентов этих систем. Формальные методы получают все большее распространение, поскольку постоянно растет спрос на надежные системы и увеличивается число специалистов, знакомых с этими методами. Но должно пройти еще немало времени, чтобы использование формальных методов в процессе разработки критических систем стало обычным делом.

21.2. Аттестация безотказности

Как уже отмечалось в главе 17, для специфицирования требований безотказности систем разработано множество различных числовых показателей. Чтобы быть уверенным, что система соответствует требованиям, необходимо измерить ее показатели безотказности, учитывая работу типичного пользователя.

Процесс измерения показателей безотказности системы представлен на рис. 21.1; он состоит из четырех этапов.

1. На этапе определения операционного профиля изучаются аналогичные существующие системы. Операционный профиль задает разные классы входных данных системы и вероятность их ввода в нормальных режимах ее работы. Этот этап рассматривается в следующем разделе.
2. Подбирается множество тестовых данных (иногда с помощью генератора тестовых данных), соответствующих операционному профилю.
3. Проводится тестирование системы с подобранными данными и подсчитывается число отказов. Также фиксируется количество повторов этих отказов. Как отмечалось в главе 17, выбранные временные единицы должны подходить для используемых показателей.
4. После получения статистически значимого количества отказов следует этап вычисления показателей безотказности системы.

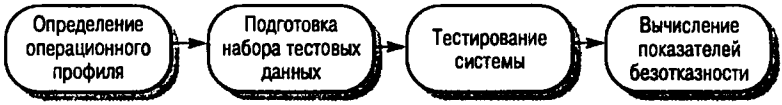


Рис. 21.1. Процесс измерения показателей безотказности

Данный метод иногда называют статистическим тестированием. Цель статистического тестирования — оценить безотказность системы [284]. Статистическое тестирование противоположно тестированию дефектов, проводимому в целях обнаружения ошибок в системе.

Этот концептуально привлекательный метод измерения безотказности не так прост для применения на практике. Принципиальные трудности возникают по нескольким причинам.

1. *Неопределенность операционного профиля.* Профили могут неточно отражать реальное использование системы.
2. *Высокая стоимость генерации тестовых данных.* Если нет возможности автоматической генерации тестовых данных, то создание большого количества тестовых данных занимает много времени.
3. *Статистическая неопределенность в случае высокой безотказности.* Для точного измерения показателей безотказности необходимо сгенерировать статистически значимое число отказов.

Конечно, разработка точного операционного профиля возможна только для систем, имеющих стандартный шаблон использования. Что касается других систем, то здесь каждый пользователь может обращаться с системой по-своему. Как уже отмечалось в главе 16, разные пользователи могут иметь различные взгляды на безотказность системы именно потому, что они используют разные ее функции и средства.

Не самым лучшим методом генерации большого множества данных, используемого для измерения безотказности, оказывается применение некоторых видов генераторов тестовых данных, которые могут автоматически создавать входные данные, соответствующие операционным профилям. Как правило, невозможно автоматизировать создание всех необходимых тестовых данных для интерактивных систем. Наборы данных для них должны создаваться вручную, а следовательно, с более высокими затратами.

Статистическая неопределенность – одна из основных проблем при измерении безотказности системы. Чтобы оценка показателей безотказности была наиболее точной, недостаточно просто спровоцировать в системе один сбой. Для точной оценки безотказности необходимо сгенерировать статистически значимое количество сбоев. В такой ситуации задача оценивания безотказности сводится в лучшем случае к минимизации количества ошибок в системе, в худшем – к измерению эффективности метода минимизации ошибок. Если в спецификации определен очень высокий уровень безотказности, то, как правило, генерация достаточного количества отказов в системе становится экономически невыгодной.

21.2.1. Операционные профили

Операционный профиль отражает практику использования системы. Он состоит из спецификации классов входных данных и вероятности их появления. В тех случаях, когда новая система ПО устанавливается вместо уже существующей, вероятный шаблон использования новой системы построить сравнительно легко. Он должен соответствовать "обычному" использованию старой системы плюс дополнительные возможности, обусловленные наличием новых функций, включенных в новое ПО. Например, нетрудно по-

строить операционный профиль для телекоммуникационных систем коммутации, поскольку телекоммуникационным компаниям известны шаблоны вызовов, которые обрабатываются в этих системах.

Обычно в операционном профиле входные данные с высокой вероятностью ввода разделяются на несколько небольших классов, на рис. 21.2 они показаны слева, а справа расположены классы входных данных, ввод которых мало вероятен, но возможен. Этих классов обычно очень много.

Рис. 21.2. Операционный профиль

В работах [246, 247] предложены правила разработки операционных профилей. Эти правила основаны на опыте построения операционных профилей для систем телекоммуникаций. Здесь накоплен достаточный опыт использования таких систем, поэтому процесс разработки операционных профилей относительно прост. Для систем, имеющих около 15 человеко-лет развития, операционный профиль был разработан примерно за 1 человеко-месяц. В других случаях создание операционного профиля заняло больше времени (2–3 человеко-года), однако, расходы на него полностью окупились после выпуска последующих систем. В этих работах подсчитано, что телекоммуникационная компания получила по крайней мере 10-кратное возмещение от инвестиций, вложенных в разработку операционных профилей.

Если система ПО инновационная, предвидеть, как она будет использоваться, намного сложнее. Система используется различными группами пользователей с разными ожиданиями, знаниями и опытом. У новых систем нет предыстории использования, и для работы с ними пользователи часто применяют способы, не предусмотренные разработчиками системы. Еще одна проблема состоит в том, что операционный профиль может меняться в ходе использования системы. По мере накопления опыта работы с системой навыки и уверенность пользователей меняются, и со временем они начинают использовать ПО более изощренными способами. Все эти причины часто не позволяют разработать надежный операционный профиль [152]. В подобных ситуациях сложно оценить степень неопределенности в измерении показателей безотказности систем.

21.2.2. Оценивание безотказности

Во время аттестации ПО менеджеры должны уделить основное внимание тестированию системы. Так как тестирование — очень дорогой процесс, важно завершить его как можно раньше, причем так, чтобы впоследствии не пришлось тестировать систему по-

вторно. Тестирование завершается, если достигнут необходимый уровень безотказности системы. Конечно, иногда выясняется, что требующийся уровень безотказности никогда не будет достигнут. В этом случае менеджер должен принять нелегкое решение о переработке некоторых частей системы или перезаключении контракта с заказчиком.

Модель возрастания безотказности[2] показывает, как меняется безотказность системы в процессе тестирования. Ошибки исправляются по мере обнаружения, так что во время тестирования и отладки системы ее безотказность должна возрастать. Для практического оценивания безотказности концептуальную модель возрастания безотказности необходимо преобразовать в математическую. Здесь не будет подробно рассматриваться такое преобразование, обсудим только саму модель возрастания безотказности.

Во многих предметных областях на основании экспериментов по оцениванию безотказности создается множество моделей возрастания безотказиости. Самая простая – это модель ступенчатой функции [190], в которой при обнаружении и исправлении ошибки безотказность каждый раз возрастает на постоянную величину (рис. 21.3). В этой модели предполагается, что ошибки всегда исправляются без внесения новых ошибок, поэтому количество отказов и связанных с ними ошибок со временем уменьшается. Соответственно, после исправлений ошибок интенсивность отказов должна понижаться, как показано на рис. 21.3. Заметим, что интервалы времени на горизонтальной оси отражают время между повторными тестированиями системы (после исправления обнаруженных ошибок), поэтому они, как правило, не равны друг другу.

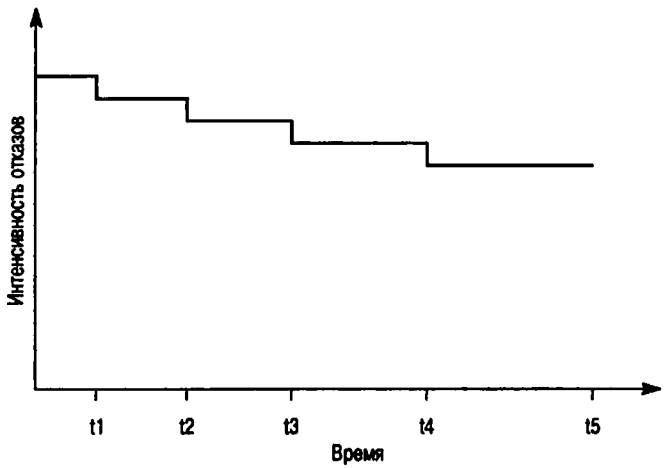


Рис. 21.3. Модель возрастания безотказности на основе равноступенчатой функции

На самом деле в процессе отладки не всегда удается локализовать ошибки; кроме того, изменение кода может привести к появлению новых ошибок. Вероятность появления новых ошибок может превысить вероятность проявления уже исправленных ошибок, в итоге безотказность системы может не возрасти, а наоборот уменьшиться.

В простой модели равноступенчатого возрастания безотказности также предполагается, что все ошибки вносят равный вклад в безотказность, поэтому каждое исправление

[2] *В русской научной литературе эту модель (точнее, семейство моделей) часто называют также моделью роста надежности. Оба названия являются переводом с английского reliability growth model.* – Прим. перев.

ошибки повышает безотказность на некоторую постоянную величину. Однако не все ошибки равновероятны. Исправление наиболее часто проявляющихся ошибок приводит к возрастанию безотказности на бо́льшую величину, чем исправление редко проявляющихся ошибок. Поэтому возрастание безотказности на каждом временном шаге не будет постоянным.

В более поздних моделях возрастания безотказности, например предложенных в работе [222], для решения этой проблемы вводится случайный элемент, который позволяет более точно оценить рост безотказности после исправлений ошибок в системе. Таким образом, при каждом исправлении ошибки безотказность возрастает не равномерно, а на некоторую величину, зависящую от случайного параметра (рис. 21.4).

Рис. 21.4. Модель возрастания безотказности на основе функции со случайным шагом

В модели, предложенной в [222], возможно уменьшение безотказности, если исправление ошибок вносит в ПО дополнительные ошибки. Данная модель также демонстрирует тот факт, что по мере исправления ошибок скорость возрастания безотказности в среднем понижается. Причина такого понижения обусловлена тем, что в процессе тестирования наиболее вероятные ошибки обнаруживаются самыми первыми, при этом исправление наиболее вероятных ошибок вносит наибольший вклад в возрастание безотказности.

Описанные выше модели являются дискретными и основаны на пошаговом возрастании безотказности. Перед тестированием новая версия ПО с исправленными ошибками должна иметь более низкую интенсивность отказов, чем предыдущая. Но для того, чтобы количественно оценить безотказность системы после заданного количества тестирований, нужны математические модели. Множество таких моделей, адаптированных для разных предметных областей представлено в работах [2, 246, 26*].

Проще всего оценить безотказность, подобрав к измеренным данным известную модель безотказности. Затем эта модель экстраполируется на требующийся уровень безотказности. Время, за которое можно достичь заданной безотказности, затем легко определить по графику модели безотказности (рис. 21.5). Тестирование и отладка системы должны продолжаться до момента времени достижения необходимого уровня безотказности.

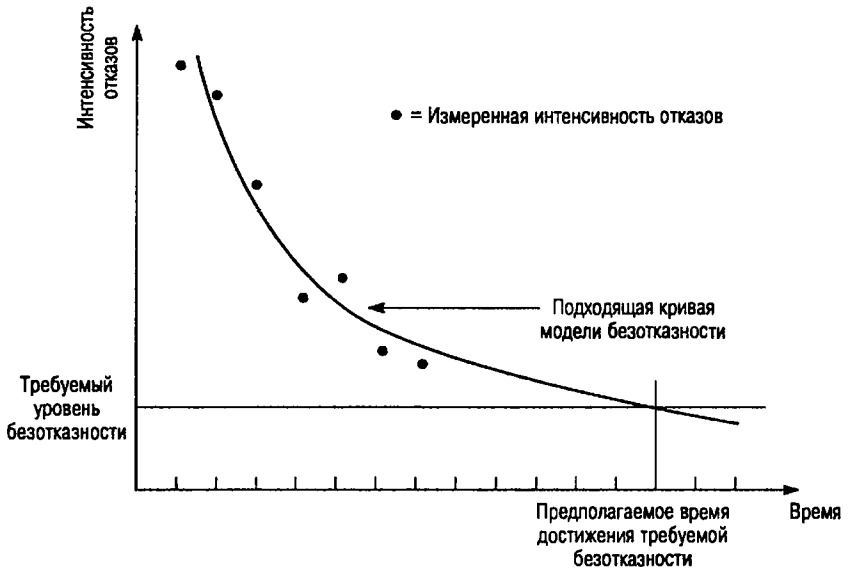


Рис. 21.5. Оценивание безотказности

Оценка безотказности системы по модели безотказности имеет два принципиальных преимущества.

1. *Планирование тестирования.* При составлении графика тестирования можно оценить время, в течение которого завершится тестирование. Если это время оказывается больше, чем запланировано в графике разработки системы, для тестирования необходимо привлечь дополнительные ресурсы.
2. *Требования заказчика.* Иногда модель безотказности показывает, что возрастание безотказности идет медленно и для относительно малого прироста безотказности требуется приложить непропорционально много усилий. В такой ситуации совместно с заказчиком можно пересмотреть требования надежности. В другой ситуации модель может показать, что необходимый уровень безотказности, по всей видимости, никогда не будет достигнут. В этом случае также следует пересмотреть системные требования.

Использование моделей возрастания безотказности и их применение на практике описано в работах [105, 307, 312, 29*].

21.3. Гарантии безопасности

Получение гарантий безопасности системы и аттестация ее безотказности — разные процессы. Безотказность можно определить количественно с помощью различных числовых показателей. Безопасность нельзя достоверно определить каким-либо количественным способом и, следовательно, невозможно измерить в ходе тестирования системы.

Поэтому аттестация безопасности определяет уровень надежности системы, который может варьироваться от "очень низкого" до "очень высокого". Здесь требуется профессиональная оценка безопасности. Во многих случаях определение безопасности частично базируется на опыте организации, разрабатывающей систему. Если в организации уже

есть предварительно разработанные надежно функционирующие безопасные системы, то разумно предположить, что в данной организации будут разработаны подобные безопасные системы. С другой стороны, оценка безопасности должна опираться на реальную архитектуру системы, результаты верификации и аттестации, а также на процессы, которые применялись при разработке системы.

21.3.1. Верификация и аттестация

Верификация и аттестация систем, критических по обеспечению безопасности, имеет много общего с тестированием любых систем с высокими требованиями надежности. Чтобы обнаружить как можно больше системных дефектов, следует применить всестороннее тестирование, а при оценке безотказности системы использовать статистическое тестирование. Однако вследствие чрезвычайно низкой частоты отказов, присущих многим системам, критическим по обеспечению безопасности, с помощью статистического тестирования не всегда удается количественно оценить безотказность системы, поскольку для этого требуется провести чрезвычайно большое количество тестов. При оценке безопасности системы эти тесты, используемые совместно с такими методами тестирования, как инспектирование кода и статические проверки (см. главу 19), всего лишь дают основание считать систему безопасной.

При создании систем, критических по обеспечению безопасности, важен всесторонний анализ разрабатываемой системы. Предложено пять типов анализа системы, обязательных для систем с критическими требованиями безопасности [270].

1. Анализ правильности функционирования системы.
2. Анализ возможности изменения и понятности системной архитектуры.
3. Анализ соответствия алгоритма обработки и структуры данных определенному в спецификации поведению системы.
4. Анализ согласованности программного кода, алгоритмов и структур данных.
5. Анализ адекватности тестовых сценариев системным требованиям.

Все доказательства безопасности системы строятся на следующем предположении: количество ошибок в системе, которые приводят к аварийным ситуациям, намного меньше общего количества имеющихся в системе ошибок. Обеспечение безопасности должно сосредоточиться на выявлении потенциально опасных ошибок. Если оказывается, что эти ошибки не проявляются или проявляются, но не приводят к серьезным последствиям, то система считается надежной. В следующем разделе обсуждаются основы доказательства безопасности.

21.3.2. Доказательство безопасности

Доказательства правильности программ были предложены в качестве методов верификации ПО более 25 лет назад. Однако эти методы используются в основном в исследовательскнх лабораториях. Практические проблемы построения доказательства правильности ПО настолько сложны, что некоторые организации считают использование данных методов в процессе разработки обычных систем неоправданно дорогим. Но, как подчеркивалось в начале главы, для ряда критических приложений экономически выгодно использовать доказательства правильности системы, чем ликвидировать последствия отказов.

Несмотря на то что для большинства систем разрабатывать доказательства правильности нерентабельно, иногда возникает необходимость разработать доказательства безопасности, демонстрирующие соответствие данной программы требованиям по обеспече-

нию безопасности. При доказательстве безопасности необязательно доказывать соответствие программы спецификации. Необходимо только показать, что выполнение программы не приводит к сбоям с опасными последствиями.

Наиболее эффективный метод доказательства безопасности системы – доказательство от противного. В начале доказательства делают предположение, что во время работы программы может возникнуть опасное состояние, определенное в процессе анализа рисков. Затем выполняют систематический анализ кода и доказывают, что предусловия для данного состояния противоречат постусловиям всех ветвей программы, приводящих к данному состоянию. Если это действительно так, то начальное предположение об опасном состоянии неверно. Если это же верно для всех других определенных рисков, значит, система безопасна.

Рассмотрим для примера код, представленный в листинге 21.1, который является частью реализации системы управления инъекциями инсулина. В код добавлены комментарии, чтобы связать его с деревом отказов, показанным на рис. 17.3.

Листинг 21.1. Программа управления инъекциями инсулина

```
//Вводимая доза инсулина является функцией уровня сахара в крови,
//предыдущей суммарной введенной дозы и времени предыдущей инъекции
    currentDose = computeInsulin () ;
    //Проверка безопасности: если необходимо, изменяется currentDose
    //1-й оператор if
    if (previousDose == 0)
    {
        if(currentDose > 16)
            currentDose = 16;
    }
    else
        if (currentDose > (previousDose * 2) )
            currentDose = (previousDose * 2;
    //2-й оператор if
    if (currentDose < minimumDose)
        currentDose = 0;
    else if (currentDose > maxDose)
        currentDose = maxDose;
    administerInsulin (currentDose);
```

Доказательство безопасности для данного кода включает доказательство того, что вводимая доза инсулина никогда не превысит максимальный уровень, отдельно установленный для каждого больного диабетом. Таким образом, совсем необязательно доказывать, что система выдает "правильную" дозу, достаточно доказать, что пациенту никогда не будет выдана слишком большая доза инсулина.

Для построения доказательства безопасности находим предусловие для опасного состояния, в данном случае это условие currentDose > maxDose. Затем необходимо показать, что все ветви программы приводят к противоречию с данным условием. Если это действительно так, то условие опасного состояния никогда не будет истинным, а следовательно, система безопасна.

Доказательство безопасности, подобное представленному на рис. 21.6, намного короче, чем формальная верификация системы. Сначала определяются все возможные ветви, которые приводят к потенциально опасному состоянию. Для этого необходимо пройти в обратном порядке все ветви, идущие от данного опасного состояния, и рассмотреть последнее присвоение у всех переменных состояния на каждой ветви. Предыдущие вычис-

ления (например, 1-й оператор if на рис. 21.6) можно не рассматривать. В этом примере нам необходимо рассмотреть набор возможных значений переменной currentDose (текущая доза) непосредственно перед выполнением метода administerInsulin (регулирование инсулина).

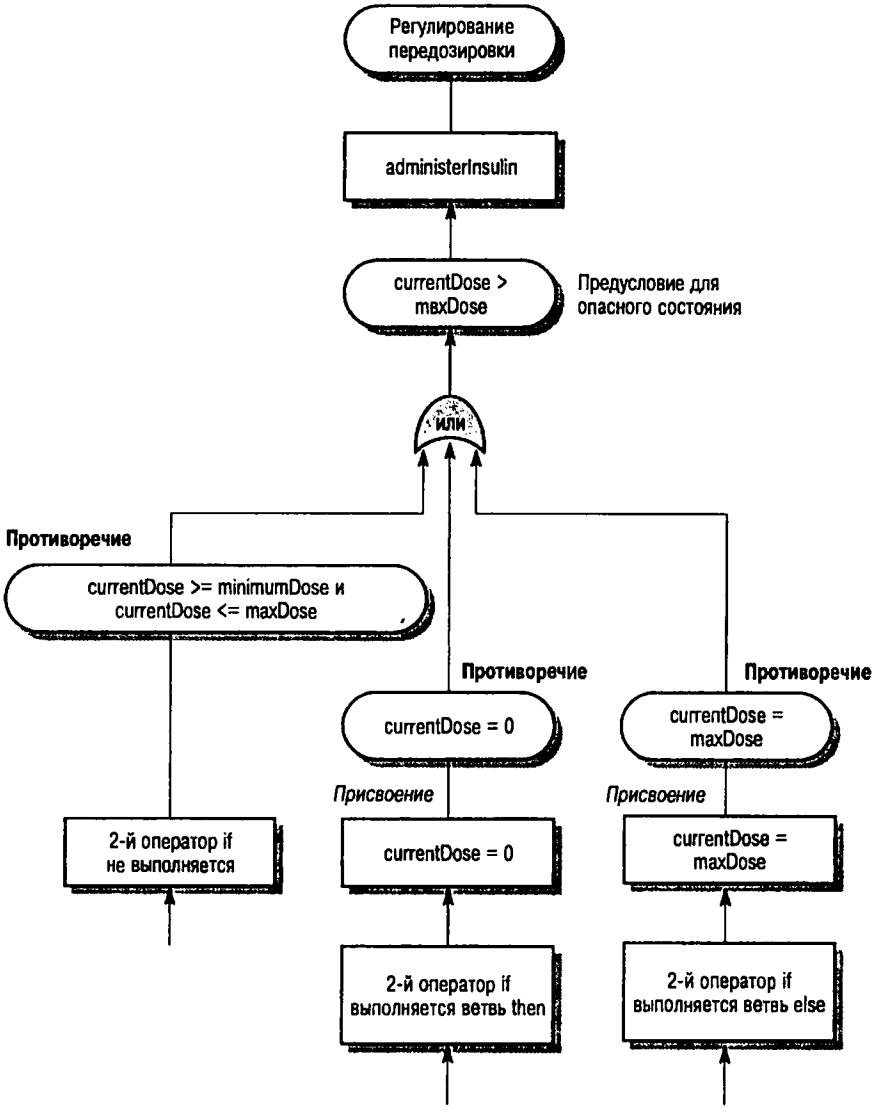


Рис. 21.6. Доказательство безопасности методом от противного

В представленном на рис. 21.6 доказательстве вызов метода administerInsulin возможен в трех ветвях программы.

1. Ни одно из ответвлений 2-го оператора if не выполняется. Данная ситуация имеет место, если значение переменной currentDose больше или равно minimumDose (минимальная доза) и меньше или равно maxDose (максимальная доза).
2. Выполняется ветвь then 2-го оператора if. В этом случае currentDose присваивается значение нуль. Здесь постусловием будет currentDose = 0.
3. Выполняется ветвь else 2-го оператора if. В этом случае currentDose присваивается значение maxDose. Постусловием этой ветви будет currentDose = maxDose.

Во всех трех случаях постусловие противоречит предусловию опасного состояния, поэтому система безопасна.

21.3.3. Обеспечение безопасности в процессе разработки ПО

В начале главы уже поднимался вопрос о важности обеспечения качества процесса разработки системы. Данный вопрос важен для любых критических систем, а особенно для систем, критических по обеспечению безопасности, на что есть две причины.

1. Аварийные ситуации — редкие события в критических системах, поэтому практически невозможно смоделировать их во время тестирования системы.
2. Требования безопасности, как указывалось в главе 16, иногда не исключают ненадежного поведения системы. Посредством тестирования и других процессов аттестации невозможно полностью доказать соответствие системы требованиям безопасности.

Из модели жизненного цикла разработки систем с критическими требованиями безопасности следует, что безопасности необходимо уделять внимание на протяжении всех этапов процесса разработки ПО. Поэтому в процесс разработки критической системы необходимо включить мероприятия по обеспечению безопасности разрабатываемой системы и среди них перечисленные ниже.

1. Создание системы регистрации и наблюдения, которая отслеживает опасности в системе, начиная с этапа предварительного анализа рисков и заканчивая тестированием и аттестацией системы.
2. Назначение инженеров, которые будут отвечать за все аспекты безопасности системы.
3. Всесторонний анализ безопасности на протяжении всего процесса разработки.
4. Создание системы сертификации безопасности, посредством которой аттестуются компоненты с критическими требованиями безопасности.
5. Использование четко разработанной системы управления конфигурацией (см. главу 29), которая необходима для отслеживания всех связанных с системной безопасностью документов и их согласования с соответствующей технической документацией.

Чтобы показать, что представляет собой процесс аттестации систем, критических по обеспечению безопасности, рассмотрим процесс анализа рисков, который является важной частью разработки таких систем. Напомним, что в процессе анализа рисков, рассмотренного в главе 17, определяются и анализируются системные риски. Анализ рисков выполняется на протяжении всего процесса разработки системы, поэтому необходимо быть уверенным, что на всех этапах процесса разработки риски рассмотрены должным образом.

Если в процессе разработки четко определены внутренние риски системы, то далее доказывается, что эти риски не могут привести к аварийным ситуациям. Это доказательство может быть подкреплено доказательством безопасности, рассмотренным в разделе 21.3.2. Основным документом безопасности является документ, в котором проанализированы и отслежены все риски, определенные в процессе специфицирования системы. Этот документ затем используется на всех этапах процесса разработки ПО для оценки того, как на каждом этапе разработки учитываются риски. Упрощенный пример анализа рисков, выполненный для системы инъекции инсулина, показан во врезке 21.1.

Врезка 21.1. Анализ рисков

Анализ рисков	**Стр. 4. Печать 20.02.99.**
Система: Система инъекций инсулина	*Файл: InsulinPump/Safety/HazardLog*
Инженер по безопасности: Джеймс Браун	*Версия анализа: 1/3*
Опасное событие	Передозировка инсулина
Обнаружил	Джейн Вильямс
Класс критичности	1
Степень риска	Высокая
Дерево отказов определено	ДА. *Дата* 24.01.99. *Документ* Анализ сбоев, стр. 5
Дерево отказов построено	Джейн Вильямс и Билл Смит
Дерево отказов проверено	ДА. *Дата* 28.01.99. *Проверил* Джеймс Браун

Требования безопасности системы

1. В системе должны быть средства для самотестирования системы датчиков, часов и средств инъекции инсулина.
2. Самопроверка ПО должна выполняться один раз в минуту.
3. Если при проверке ПО обнаруживаются ошибки в каких-либо компонентах системы, то выдается звуковое предупреждение, а на мониторе отображается название компонента, в котором обнаружены ошибки. Инъекция инсулина при этом приостанавливается.
4. В системе должна быть возможность ручной корректировки дозы, которая позволяет пользователю изменять рассчитанную ранее дозу инсулина.
5. Новое значение дозы должно быть не больше установленного предельного значения. Таких предельных значений может быть несколько, если система конфигурируется медицинским персоналом.

Как следует из приведенного примера анализа рисков, желательно назначить ответственное лицо (специалиста), который отвечал бы за безопасность будущей системы, но не участвовал в ее разработке. Задача такого специалиста – следить, чтобы все необходимые мероприятия для обеспечения безопасности были выполнены и описаны в соответствующих документах. Заказчик системы может также потребовать независимого проверяющего по обеспечению безопасности из сторонней организации, который будет отчитываться непосредственно перед заказчиком.

В большинстве случаев отвечать за обеспечение безопасности должны дипломированные специалисты. В Великобритании такими специалистами, как правило, являются со-

трудники проектных институтов и квалифицированные инженеры. За обеспечение безопасности не должны отвечать малоопытные некомпетентные инженеры. Не следует привлекать для этого и специалистов по программному обеспечению. Возможно, в будущем, в соответствии с более развитыми стандартами процесса разработки критического ПО, потребуется, чтобы инженеры, отвечающие за обеспечение безопасности, были дипломированными специалистами соответствующего профиля.

21.3.4. Контроль безопасности исполнения

В главе 18 вы познакомились с понятием "безопасное программирование". В отличие от обычного программирования, при безопасном в программу добавляются дополнительные операторы для проверки наличия ошибок в системе. Подобная технология может использоваться и в системах, критических по обеспечению безопасности. В программу добавляется контрольный код, который проверяет ограничения безопасности, и в случае их нарушения включается обработка исключительной ситуации. Ограничения безопасности можно представить в виде формальных условий. Эти условия являются предикатами, описывающими ограничения, которые должны выполняться перед следующим исполняемым оператором. В системах с критическими требованиями безопасности, формальные условия должны создаваться на основе спецификации требований безопасности. Такие ограничения безопасности более полно обеспечивают надежное поведение системы.

Особое значение имеют условия для обеспечения безопасности взаимодействий компонентов системы. Например, в системе инъекции инсулина при регулировании дозы в блок инъекции может поступить сигнал на увеличенную дозу инсулина (листинг 21.2). Максимально допустимое увеличение дозы инсулина можно определить заранее и включить в систему в виде формального условия.

Листинг 21.2. Управление инъекцией инсулина

```
static void administerInsulin ( ) throws SafetyException {
    int maxIncrements = InsulinPump.maxDose / 8;
    int increments = InsulinPump.currentDose / 8;
    // проверка условия currentDose <= InsulinPump.maxDose
    if (InsulinPump.currentDose > InsulinPump.maxDose)
        throw new SafetyException (Pump.doseHigh);
    else
        for (int i = 1 ; i <= increments; i++)
        {
            generateSignal ( );
            if (i > maxIncrements)
                throw new SafetyException (Pump.incorrectIncrements) ;
        }
}// administerInsulin
```

Если при вычислении значения переменной currentDose (текущая доза) возникает ошибка, то происходит прерывание программы. Сигнал на чрезмерную дозу инсулина не выдается, так как контрольное условие гарантирует, что система выдаст дозу, не большую, чем maxDose (максимальная доза). В принципе большая часть кода с условиями безопасности может быть создана автоматически с помощью препроцессора обработки условий. Однако такой способ не всегда приемлем, часто код контрольных условий создается вручную.

21.4. Оценивание защищенности ПО

В настоящее время оценивание защищенности системы приобретает большое значение, поскольку все чаще системы объединяются посредством Internet. Как уже отмечалось в главе 16, требования защищенности в некоторых отношениях подобны требованиям безопасности. В частности, они определяют нештатное поведение системы, а не ее "рабочее" поведение. Однако, как правило, невозможно определить это поведение в виде простых ограничений, контролируемых системой.

Основное отличие между безопасностью и защищенностью состоит в том, что проблемы безопасности обычно имеют случайный характер, в то время как атаки на защищенность системы совершаются преднамеренно. Даже в тех системах, которые используются уже много лет, изобретательный взломщик может найти новый способ проникнуть в систему, которая до сих пор считалась защищенной. Например, долгое время считавшийся безопасным алгоритм шифрования данных RSA был взломан в 1999 году.

Существует четыре дополняющих друг друга метода проверки защищенности.

1. *Аттестация, основанная на предыдущем опыте.* Здесь система анализируется по отношению к различным типам атак, известных группе аттестации. Данный тип аттестации обычно проводится совместно с аттестацией, выполняемой инструментальными средствами.
2. *Аттестация инструментальными средствами.* Здесь для анализа системы используются разные инструменты защищенности, например программа проверки паролей. Данный метод дополняет аттестацию, основанную на предыдущем опыте.
3. *Команда взлома.* Команда ставит перед собой цель — взломать систему разными способами. Например, моделируются атаки на систему.
4. *Формальная верификация.* Выполняется проверка системы на соответствие формальной спецификации защищенности. Этот метод не имеет широкого применения.

Конечным пользователям очень сложно проверить защищенность системы. Поэтому в Северной Америке и Европе выработаны системы критериев оценки защищенности, которые контролируются специально обученными экспертами [131]. Поставщики готового ПО могут представить на рассмотрение свои конечные продукты для оценки и сертификации по различным критериям защищенности.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Из-за высокой стоимости последствий отказов в критических системах такие методы верификации и аттестации, как формальная спецификация и доказательства, требующие значительных расходов, могут оказаться рентабельными для критических систем.
- Статистическое тестирование применяется для оценки безотказности ПО. При этом проводится тестирование системы с наборами тестовых данных, соответствующими операционным профилям данного ПО. Генерация тестовых данных может быть автоматизирована.
- Модели возрастания безотказности моделируют изменения безотказности в процессе тестирования по мере исправления ошибок в ПО. Модели возрастания безотказности можно использовать для оценивания момента достижения необходимой безотказности и завершения тестирования ПО.

- Доказательство безопасности является эффективным методом оценивания надежности готового продукта. В ходе доказательства демонстрируется невозможность возникновения отказов в системе. Обычно доказательство безопасности проще, чем доказательство соответствия программы своей спецификации.
- Аттестацию защищенности можно провести методом анализа, основанного на предыдущем опыте, с помощью инструментальных средств или с помощью команды, имитирующей атаки на систему.

Упражнения

21.1. Проведите аттестацию по спецификации безотказности для программной системы, управляющей сетью кассовых терминалов в магазине, описанной в упражнении 17.3. Опишите все методы и средства, которые использовались в процессе аттестации.

21.2. Объясните, почему практически невозможно проверить безотказность системы, если требование безотказности определено как малое число отказов за время работы системы.

21.3. Этично ли поведение разработчика, который соглашается поставить заказчику программную систему с известными дефектами? Этично ли его поведение, если в данной ситуации он заблаговременно предупредит заказчика о наличии в системе ошибок? Разумно ли заявлять о безотказности системы в таких обстоятельствах?

21.4. Объясните, почему обеспечение безотказности системы не гарантирует безопасность системы.

21.5. Система управления дверным замком в системе хранения радиоактивных отходов спроектирована с учетом безопасности обслуживающего персонала. Вход в хранилище разрешен, только когда установлены защитные экраны или если уровень радиации в хранилище ниже некоторого установленного значения (dangerLevel). Итак:

- если контролируемые на расстоянии защитные экраны установлены, зарегистрированный оператор может открыть дверь;
- если уровень радиоактивности меньше некоторого определенного значения, зарегистрированный оператор может открыть дверь;
- регистрация оператора определяется посредством ввода им личного входного кода.

Программный код на языке Java, используемый для управления механизмом дверного замка, представлен в листинге 21.3. Заметим, что безопасное состояние состоит в том, чтобы не разрешить вход в хранилище, если не выполнены перечисленные выше условия.

Покажите, что код в листинге 21.3 потенциально небезопасный. Измените его так, чтобы он стал безопасным.

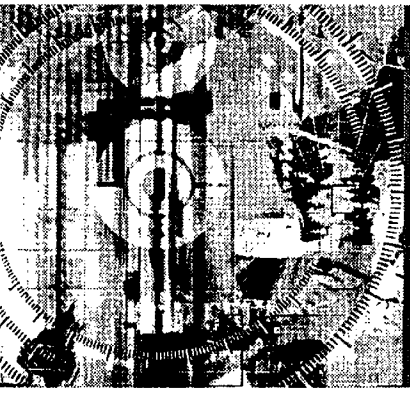
21.6. Используя спецификацию вычисления дозы в системе управления инъекцией инсулина (см. главу 9, рис. 9.10), по аналогии с программой из листинга 21.1 напишите на языке Java код метода computeInsulin. Постройте неформальное доказательство безопасности этого кода.

21.7. Опишите, как вы проводите аттестацию системы защиты с помощью паролей для разработанных вами приложений. Объясните назначение и применение любого инструментального средства, которое, на ваш взгляд, может оказаться полезным.

21.8. Предположим, что вы представитель группы разработчиков программного обеспечения для химического завода, работа которого вызывает серьезное загрязнение окружающей среды. Ваш руководитель во время интервью на телевидении заявил, что процесс аттестации программной системы всесторонний и ошибок в ней быть не может. Далее он сообщил, что проблемы возникают вследствие неправильных действий операторов, обслуживающих систему. Системные операторы не согласны с такими заявлениями и предлагают бастовать. Как вы будете действовать в сложившейся ситуации?

Листинг 21.3. Управление дверным замком

```
entryCode = lock.getEntryCode();
if(entryCode == lock.authorisedCode)
{
    shieldStatus = Shield.getStatus();
    radiationLevel = RadSensor.get();
    if(radiationLevel < dangerLevel)
        state = safe ;
    else
        state = unsafe ;
    if(shieldStatus == Shield.inPlace())
        state = safe ;
    if(state == safe)
        {
            Door.locked = false ;
            Door.unlock();
        }
    else
    {
        Door.lock();
        Door.locked = true;
    }
}
```



Часть VI

Управление

22 Управление персоналом

Цели

Цель этой главы – рассмотреть роль человеческого фактора в процессе разработки программного обеспечения. Изучив материал главы, вы должны:

- знать основные модели организации человеческой памяти, модели решения проблем и мотивации, а также механизмы их применения в практической работе руководителей проектами по созданию программного обеспечения;
- иметь представление об основных проблемах командной работы (в частности, при подборе членов команды), о сплоченности команды, внутрикомандных взаимоотношениях и ее структуре;
- знать о проблемах, связанных с подбором и сохранением персонала в организации, занимающейся разработкой и сопровождением программного обеспечения;
- иметь понятие о модели P-CMM, которая является основой для повышения производительности сотрудников организации, занимающейся созданием программного обеспечения.

Содержание

Люди, работающие в компаниях по разработке ПО, являются их самым ценным "активом". Именно они представляют интеллектуальный капитал, и от менеджеров по разработке ПО зависит, получит ли компания наилучшие из возможных дивиденды от инвестиций в человеческие ресурсы. В успешно развивающихся компаниях и экономических структурах это достигается в том случае, если организация уважает своих сотрудников. Круг выполняемых ими обязанностей и уровень вознаграждения должны соответствовать их умению, которое, в свою очередь, зависит от квалификации.

Таким образом, эффективный менеджмент — это эффективное управление персоналом компании. Менеджеры — руководители проектов должны решать как технические, так и далекие от технических вопросы, используя при этом членов команды с оптимальной эффективностью. Они должны уметь мотивировать действия людей, организовывать их работу и гарантировать ее качественное исполнение. Слабый менеджмент персонала — одна из основных составляющих провала программных проектов.

В своих рассуждениях я опираюсь в основном не на принятые в наше время модные модели теории менеджмента, а на два фактора — познавательный и социальный. Процесс разработки ПО включает в себя познавательный и социальный процессы, поэтому именно эти факторы я считаю самыми важными в углублении понимания того, как создаются программы. Если у менеджеров есть понятие об этих основах, тогда они лучше управляют людьми, получая наибольшую отдачу.

22.1. Пределы мышления

Человеческие возможности достаточно разнообразны, и это прежде всего зависит от уровня интеллекта, образования и личного опыта, однако у всех нас есть нечто общее: в своей умственной деятельности мы подчиняемся определенным ограничениям. Эти ограничения не что иное, как следствие того, каким образом наш мозг сохраняет и обрабатывает полученную информацию. Нам вовсе не обязательно подробно изучать процесс обработки информации об окружающем мире человеческим мозгом. Однако я считаю, что знание границ нашего мышления имеет исключительную важность. Именно эти знания помогут выяснить, почему некоторые технологии разработки программного обеспечения достигают высокой эффективности, а также проникнуть вглубь взаимоотношений между членами команды разработчиков ПО.

22.1.1. Организация человеческой памяти

Программные системы представляют собой абстрактные объекты, при работе с которыми разработчикам необходимо учитывать все их особенности. К примеру, программист должен понимать и всегда помнить о взаимосвязи между листингом исходного кода и динамическим поведением программы. Эти знания будут полезны ему и при разработке программ в дальнейшем.

Организацию человеческой памяти можно представить в виде иерархической структуры с тремя отчетливыми, связанными между собой типами памяти (рис. 22.1).

1. *Кратковременная память с быстрым доступом, но ограниченными возможностями.* Вводный сигнал, порождаемый органами чувств, поступает именно сюда для дальнейшей обработки. Такую память мы можем сравнить с регистрами в вычислительной машине, так как она является местом обработки информации, а не ее хранения.
2. *Промежуточная память с высокими возможностями.* Доступ к такой памяти более труден, нежели к кратковременной. Она тоже служит для обработки информации, однако может удерживать информацию на более долгий срок, чем кратковременная

память. Она не используется для сохранения информации, предназначенной для более длительного хранения. По аналогии с компьютером такая память напоминает оперативную, в которой информация сохраняется только на период вычислительных операций.

3. *Долговременная память.* Это память с самыми широкими возможностями, относительно трудным доступом и крайне ненадежными механизмами хранения (мы забываем иногда некоторые вещи). Такая память используется для "постоянного" хранения информации. Продолжая наше сравнение с компьютерной памятью, можем соотнести долговременную память с дисковыми накопителями.

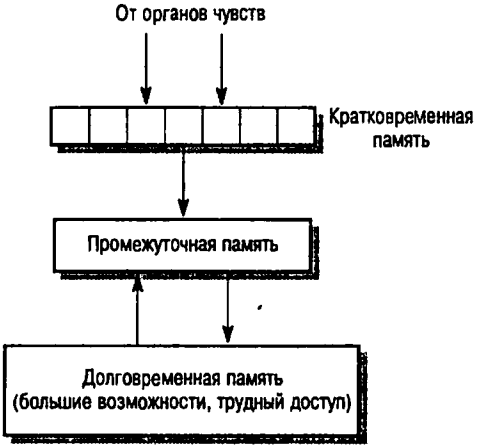


Рис. 22.1. Схема организации человеческой памяти

Кратковременная память получает информацию о поставленной перед человеком задаче через чтение (человеком) различных документов и контакты с другими людьми. Далее полученная информация, составив единое целое с уже существующими данными в долговременной памяти, поступает в промежуточную память. Именно результат формирования этого целого и составляет основу решения поставленной задачи. Все это заносится в долговременную память с тем, чтобы использоваться в будущем. Никто не говорит, что решение обязательно должно быть верным. Со временем, после поступления новой информации, долговременная память изменяется. Несмотря на это, неверная информация никоим образом не удаляется, а наоборот, сохраняется в видоизмененной форме. Ведь учимся же мы на собственных ошибках.

Наш познавательный процесс в некоторой степени сдерживается ограниченным размером кратковременной памяти. В классическом эксперименте Миллер (Miller, [237]) определил, что в кратковременной памяти может храниться около семи квантов информации. Квант информации не составляет твердо фиксированной величины, это, скорее, определенная информационная единица, которой может быть телефонный номер, сформулированная цель или название улицы. Миллер также дает описание процесса "образования блоков", во время которого кванты информации собираются в целые блоки.

Если постановка задачи содержит больше информации, чем тот объем, с которым может справиться кратковременная память, тогда эта информация будет обрабатываться и преобразовываться параллельно поступлению. Такой процесс может привести к потере информации. Также велика возможность возникновения ошибок, так как обработка информации не успевает за ее поступлением в память.

Шнейдерман (Shneiderman, [314]) предполагает, что процесс формирования блоков информации используется и при чтении программы программистом. Читающий разбивает содержащуюся в программе информацию на блоки, построенные по принципу внутренней семантической (смысловой) структуры. Восприятие программы не происходит последовательно от оператора к оператору, если только оператор не представляет собой логический блок. На рис. 22.2 показано, как простая программа сортировки может быть разбита на блоки читающим ее субъектом.

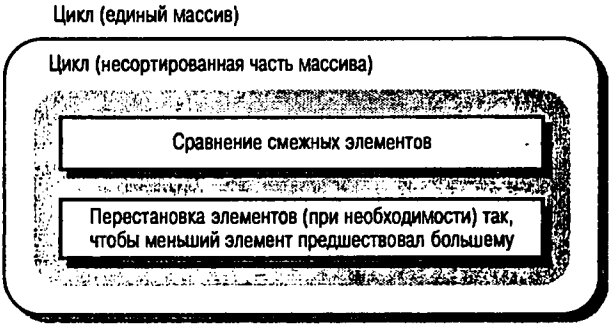


Рис. 22.2. Познавательные блоки в программе сортировки

После определения внутренней семантической структуры, представляющей программу, эти знания передаются в долговременную память. Если эта информация потребляется регулярно, то обычно ее трудно забыть. Без особого труда такую информацию можно воспроизвести в различных формах представления. Поэтому нам легче запомнить абстрактные конструкции высокого уровня, чем подробности низкого уровня.

Знания, которые приобретаются в процессе разработки ПО и сохраняются в долговременной памяти, разделяются на два класса.

1. *Семантические знания.* Это знания об основных понятиях, таких, например, как функционирование оператора присвоения, представление о классе объектов, о технике хешированного поиска или о структуре организации программ. Эти знания приобретаются через опыт и обучение и сохраняются в форме автономных представлений.
2. *Синтаксические знания.* Это детализированные знания (подробности) об отдельных объектах и явлениях, например о том, как дать описание объекта в UML, какие стандартные функции доступны в языке программирования, создается ли оператор присвоения с помощью знака "=" или знака ":=" и т.д. Эти знания хранятся в неструктурированном виде.

Такая организация знаний показана на рис. 22.3, который был заимствован (и переделан) из книги по проектированию интерфейсов пользователя [315]. Здесь предполагается, что семантические знания и знания о решаемой задаче имеют свою организацию и структуру, которые показаны на схеме в виде логической структуры взаимосвязей между различными фрагментами знаний. В противовес семантическим знаниям, синтаксические являются произвольными (в определенной мере) и не имеют четкой организации. Поэтому именно для данного типа знаний более вероятны ошибки и потеря информации.

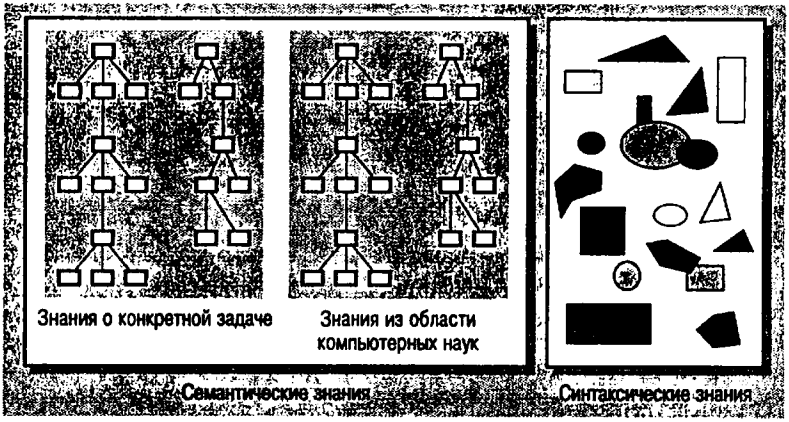


Рис. 22.3. Синтаксические и семантические знания

Семантические знания приобретаются с помощью опыта и путем активного обучения. Новая информация сознательно интегрируется с уже существующими семантическими структурами. Напротив, синтаксические знания приобретаются путем простого запоминания. Хотя новые синтаксические знания не объединяются с существующими в тот же момент, однако могут с ними взаимодействовать. Следовательно, эти знания забываются чаще, чем более углубленные семантические.

Различные модели приобретения синтаксических и семантических знаний помогают понять, как опытные программисты изучают новый язык программирования. У них не возникает особых трудностей в освоении основных понятий языка, таких, как присвоение, цикл, условные операторы и т.п. Синтаксис нового языка, тем не менее, имеет тенденцию смешиваться с синтаксисом уже знакомых языков. Поэтому программист, владеющий языком Ada, при изучении языка Java напишет оператор присвоения скорее с использованием знака ":=", чем "=".

По мере более глубокого освоения основные понятия закрепляются в памяти как семантические знания. Семантические знания сохраняются в памяти в абстрактной смысловой форме, но детали основных понятий могут быть воспроизведены в различных конкретных представлениях [67, 319].

Для примера рассмотрим алгоритм двоичного поиска, в котором осуществляется поиск конкретного элемента в упорядоченной совокупности. Этот алгоритм включает проверку средней точки совокупности и применение знаний о взаимоотношении порядка для проверки местонахождения элемента-ключа – в верхней или нижней части совокупности. Программист, знакомый с этим алгоритмом, без труда напишет его версию на языках Java, Ada или на любом другом языке программирования.

Эта модель хорошо объясняет и тот факт, что для многих людей умение программировать приходит как бы одним махом, после долгого периода обучения и трудностей. Умение программировать требует от человека понимания семантических концепций и способности разделять семантические и синтаксические понятия. Иногда преподавателям не удастся понять проблемы студентов. Сами они успешно разобрались в семантической информации и осмыслили ее, поэтому для них главная задача состоит в обработке синтаксической информации. Поэтому часто преподавателю бывает сложно объяснить семантические понятия на таком уровне, чтобы это было доступно и новичкам в программировании.

22.1.2. Решение задач

Разработка и написание программы представляет собой процесс решения задач. Для того чтобы создать систему ПО, в первую очередь необходимо понять поставленную задачу (проблему), разработать стратегию поиска решения и преобразовать решение в программу.

Первый этап включает переход постановки задачи из кратковременной памяти в промежуточную. Далее проблема сопоставляется и интегрируется с уже имеющимися знаниями в долговременной памяти, а затем обрабатывается в целях составления определенного решения. В заключение найденное решение переносится в исполняемую программу (рис. 22.4).

Рис. 22.4. Решение задачи

Процесс решения задач требует интеграции постановки задачи и компьютерных знаний. Большое значение также имеют организационные моменты, такие, как завершение работы над решением в рамках возможностей бюджета. Таким образом, пользователь должен разбираться в содержательной постановке задачи, разработчик программного обеспечения должен быть компетентным в вычислительных системах, а менеджер должен быть хорошим специалистом в организационных вопросах. В процессе разработки ПО все эти знания объединяются и используются совместно.

Разработка решения (программы) включает в себя построение внутренней семантической модели задачи и соответствующей ей модели решения. После формирования модели ее следует представить в подходящей синтаксической системе нотаций. Таким образом, создание программы представляет собой итерационный процесс, состоящий из трех этапов.

1. Интеграция существующих знаний о компьютерных технологиях и о поставленной задаче с тем, чтобы создать новое знание и с его помощью разобраться в проблеме. В работе [84] подчеркивается особая роль практического опыта решения прикладных задач на этой стадии.
2. Создание семантической модели решения, которая тестируется и совершенствуется до тех пор, пока не будет успешно справляться с поставленной задачей.
3. Представление модели на любом языке программирования или в системе проектной нотации.

Если менеджерам необходимо определить, кого включить в долгосрочный проект, в первую очередь следует оценить способность специалиста решать всеобъемлющие проблемы и его опыт работы в данной области и лишь потом его мастерство программиста. Как только приходит понимание поставленной задачи, у опытных программистов возникают приблизительно одинаковые трудности в разработке программы, независимо от того, какой при этом используется язык программирования. Несомненно, навыки программирования необходимы, и для их развития потребуется достаточно много времени (особенно это касается таких сравнительно сложных языков, как C++). Однако, исходя из своего личного опыта, могу сказать, что гораздо легче освоить определенный язык программирования, чем развить в себе способности к решению задач.

Перевод из семантической модели в программу исключает появление ошибок, если язык программирования, выбранный программистом, содержит конструкции, соответствующие самым низкоуровневым семантическим структурам. Несмотря на разнообразие, они в любом случае должны соотноситься с такими понятиями языков программирования, как операторы присвоения, циклы, условные операторы, сокрытие информации, объекты, наследование и т.п. Чем больше семантические структуры соответствуют конструкциям языков программирования, тем легче написать программу.

Таким образом, программы, написанные с помощью языков программирования высокого уровня (например, Java), содержат меньшее количество ошибок, чем созданные с помощью языка ассемблера, так как семантические структуры низкого уровня могут напрямую кодироваться в выражения языков высокого уровня. Однако при этом могут возникнуть проблемы, если четко не разделены функциональные и объектно-ориентированные аспекты исходной задачи. Проблемы такого рода появляются у компаний, которые привыкли к стандартной методике анализа (SADT[1], например), но используют при этом объектно-ориентированное проектирование и программирование.

Приведенная модель также дает объяснение тому, почему структурированное программирование (см. главу 18) является оптимальной альтернативой при организации управления программой. В основу структурированного программирования положены такие семантические понятия, как циклы и условные операторы. Кратковременная память программиста при этом редко бывает перегруженной, тем самым снижается возможность возникновения ошибок. Структурированные программы более легки для понимания, так как их можно просмотреть от начала до конца. Структуры, необходимые для формирования семантических блоков, создаются последовательно, без участия других частей программы. Таким образом, кратковременная память программиста полностью занята только одним сеансом написания кода. При этом не возникает необходимости запрашивать из промежуточной памяти информацию о других частях программы, которые взаимодействуют с создаваемым кодом.

22.1.3. Мотивация

Координация деятельности людей для выполнения определенной работы является одной из основных задач менеджеров. Маслов (Maslow, [230]) считает, что мотивация человека направлена на удовлетворение своих потребностей. Эти потребности имеют иерархическую структуру (рис. 22.5). Самый низкий уровень иерархической структуры представляют основные физиологические потребности в пище, сне, самосохранении и др. Социальные потребности связаны с необходимостью чувствовать себя частью социальной группы. Потребности

[1] *SADT (Structured Analysis and Design Technique) – метод структурного анализа и проектирования [7*, 10*]. Принадлежит к группе методов, реализующих структурный (а не объектно-ориентированный) подход к проектированию ПО.* – Прим. ред.

в оценке ассоциируются с желанием получить определенную степень уважения в обществе, а потребность в самореализации связана с развитием личности. Естественно, приоритетными в реализации являются низшие потребности (утоление голода, например), а затем уже человек сосредоточивается на более высоких потребностях.

Рис. 22.5. Иерархическая структура человеческих потребностей

Люди, работающие в организациях, которые занимаются разработкой программного обеспечения, как правило, не испытывают сильного голода или жажды и чувствуют себя в относительной безопасности в своем окружении. Таким образом, в аспекте управления этими людьми главной задачей менеджмента является удовлетворение их потребностей, связанных с оценкой, самореализацией и необходимостью быть членом определенной социальной группы.

1. Тактика удовлетворения социальных потребностей основывается на предоставлении людям возможности и времени для встреч с коллегами, а также на том, чтобы обеспечить место для таких встреч. Неформальные и легкие в использовании средства общения (например, электронная почта) с этих позиций представляют исключительную ценность.
2. Для удовлетворения потребности в оценке крайне важно дать понять людям, насколько важна их роль в организации. Открытое признание их достижений – наиболее простой и эффективный способ удовлетворения этой потребности. Кроме того, люди должны чувствовать, что их работа оплачивается на должном уровне, который определяется их знаниями и опытом.
3. Чтобы удовлетворить потребности персонала в самореализации важно предоставить каждому сотруднику определенный уровень ответственности за сделанную работу. Это достигается путем поручения им достаточно трудных задач (но ни в коем случае не невыполнимых), а также проведения обучения, в процессе которого могут развиваться их навыки.

Рассматривая мотивацию с психологической точки зрения, автор статьи [28] выделяет три типа профессионалов.

1. *Люди с целевой ориентацией*, получающие достаточно мотивации от работы, которую выполняют. К этому типу относятся "технари", мотивация которых вызвана интеллектуальными задачами по разработке программного обеспечения.

2. *Люди с самоориентацией*, мотивация которых основана на личном успехе и признании. Они заинтересованы в разработке программного обеспечения, преследуя при этом личные интересы.
3. *Люди с внешней ориентацией*, мотивация которых требует присутствия и деятельности сотрудников. Так как в наше время создание программ становится все более ориентированным на пользователя, такие люди все чаще вовлекаются в разработку программного обеспечения.

Люди с внешней ориентацией предпочитают работать в коллективе, тогда как сотрудники с самоориентацией и целевой ориентацией стремятся к работе в одиночку. Женщины принадлежат к типу людей с внешней ориентацией чаще, чем мужчины. Они также лучше действуют в качестве распространителей информации.

Мотивация каждого индивидуума включает в себя все перечисленные элементы, хотя в определенный период времени преобладает только один тип мотивации. Никто не утверждает, что личные свойства характера любого индивидуума являются постоянными. Человеку свойственно меняться. Например, у "технарей", которые не удовлетворены оплатой своего труда, может произойти переоценка ценностей, после чего личные интересы станут для них выше заинтересованности в решении технических проблем.

Пирамида мотиваций Маслова, бесспорно, хороша, однако имеет недостаток: в ее основе лежит исключительно личностная мотивация. Здесь не уделяется достаточно внимания тому факту, что люди чувствуют себя частью организации, профессиональной группы и в большинстве своем частью какой-либо культуры. Следовательно, в мотивации задействованы не только личные интересы, но и интересы этих расширенных групп. Членство в сплоченной группе является чрезвычайно высокой мотивацией для многих. Люди, выполняющие более простые задания, часто любят ходить на работу только потому, что им нравятся сотрудники, с которыми они работают, и задания, которые они выполняют.

22.2. Групповая работа

Основная часть профессионального программного обеспечения разрабатывается командами программистов (от двух и до нескольких сотен человек). Но, поскольку едва ли кто-то способен эффективно работать над одной задачей в такой большой команде, эти команды делятся еще и на подгруппы. Каждая подгруппа отвечает за определенную часть проекта и работает над одной подсистемой. При грамотном подборе группа состоит не более чем из восьми человек. В группах небольшого размера легче снизить риск возникновения проблем во взаимоотношениях между членами группы. Каждая группа должна быть обеспечена круглым столом для проведения встреч. Кроме того, члены группы имеют возможность встречаться в офисах. Для таких групп нет необходимости применять сложные структуры коммуникации.

Организация команды, которая могла бы эффективно работать над программой, является достаточно сложной задачей для менеджера. Необходимо, чтобы в команде было равное соотношение технических навыков, опыта и выражения индивидуальности. Должен отметить, что хорошо функционирующая команда — это нечто большее, чем простой набор людей с необходимым соотношением навыков. В хорошей команде присутствует дух товарищества, который мотивирует сотрудников через успехи всей команды, включая и достижение собственных целей. Поэтому менеджеры должны стимулировать деятельность, направленную непосредственно на "строительство команды", чтобы содействовать формированию чувства преданности ее интересам.

Ниже перечислены четыре основных фактора, которые в той или иной степени влияют на групповую работу.

1. *Состав команды.* Команда должна иметь правильное соотношение навыков, опыта и личностных качеств.
2. *Сплоченность команды.* Члены рабочей группы должны воспринимать себя как единую команду, а не как простую совокупность индивидуумов, работающих над одной проблемой.
3. *Общение в команде.* Между членами команды должны быть дружеские отношения.
4. *Организация команды.* Необходимо организовать команду таким образом, чтобы каждый чувствовал свою ценность и был удовлетворен своей ролью.

22.2.1. Создание команды

В начале практической деятельности для большинства специалистов по программному обеспечению основным мотивационным фактором служит работа. Поэтому в группы по разработке ПО часто включены люди, имеющие собственные соображения насчет того, как должны решаться задачи технического плана. Такой принцип подбора рабочих групп вызван постоянными жалобами на проблемы, возникающие вследствие игнорирования стандартов интерфейса, переделки системы после создания ее программного кода, ненужных нововведений в систему и т.п. Если вы хотите избежать этих проблем, то необходимое условие — правильный подбор членов рабочей группы.

Группа, в которой сотрудники дополняют друг друга, может работать намного эффективнее группы, отбор в которую проводился исключительно на основе навыков программирования. Люди, которые любят свою работу (целевая ориентация), могут стать прекрасными профессионалами. Люди с самоориентацией на наилучший результат смогут довести дело до конца. Сотрудники с внешней ориентацией успешно налаживают общение внутри группы. Кстати, я настаиваю на том, что сотрудники с внешней ориентацией — необходимое составляющее любой рабочей группы. Они настроены на общение и поэтому могут определить (и предотвратить) возникновение какого-либо напряжения или конфликтов на ранней стадии. Именно такие люди помогут разрешить личные проблемы членов команды и разногласия между ними, прежде чем те окажут влияние на всю команду.

Иногда просто невозможно организовать команду так, чтобы все ее члены взаимодополняли друг друга по личным качествам. Если такое случается, менеджер проекта обязан контролировать группу с тем, чтобы не позволить личным целям сотрудников стать выше цели организации или группы. Достичь такого контроля намного легче в том случае, если члены команды будут участвовать в каждом этапе проекта. Проявление индивидуальной инициативы будет более вероятным, если с членами группы проводить инструктаж, не упоминая той роли, которую их часть работы играет во всем проекте.

Например, возьмем инженера, который получил проект программы для кодирования и видит, что в него необходимо внести изменения, делающие его более совершенным. Если внедрять эти изменения без понимания логики исходного проекта, это может привести к конфликту с другими частями программной системы. Если вся группа включена в разработку проекта с самого начала, работник сможет оценить последствия внесения таких изменений в систему и это не приведет к возникновению конфликта с теми специалистами, которые ее проектировали.

Важное место в команде занимает лидер. Он (или она) отвечает за техническое руководство и административное управление. Лидеры группы должны быть в курсе повседнев-

ной деятельности группы, гарантируя эффективную работу команды и тесное сотрудничество с менеджерами проекта при планировании деятельности по его реализации.

Лидер – это, как правило, назначаемая должность, он подотчетен главному менеджеру проекта. Назначаемый лидер может и не быть лидером команды в прямом смысле этого слова, он ведет группу только в технических вопросах. Члены группы могут выбрать другого лидера команды. Он может лучше назначенного лидера разбираться в технических вопросах или лучше мотивировать членов группы к выполнению работы.

Иногда бывает полезным вообще разделить техническое лидерство и управление проектом. Часто случается, что компетентные в технических вопросах работники слабо разбираются в администрировании. Исполняя роль администратора, такие лидеры уже не могут вносить полноценный вклад в техническую работу команды. В этой ситуации лучше назначить администратора, который освободит лидера от ежедневных административных проблем.

Если в группу назначается лидер, который неприемлем для членов команды, это может привести к достаточно напряженной обстановке. Члены группы не будут уважать нового лидера и могут предпочесть свои интересы интересам группы. Эта проблема весьма существенна для такой быстро изменяющейся области, как инженерия ПО, где новые члены команды могут владеть более современными знаниями и опытом, чем лидеры групп, имеющие только практический опыт. Кроме того, некоторых опытных сотрудников может обидеть назначение молодого лидера, даже если он способен внести в работу новые идеи.

22.2.2. Сплоченность команды

Члены сплоченной команды привержены ее интересам больше, чем своим собственным. Люди в хорошо управляемой, сплоченной команде преданы ее интересам. У них высоко развит командный стиль и отождествление целей группы с личными интересами. Они также воспринимают группу как некую единую сущность и пытаются защитить ее от внешнего вмешательства. Это укрепляет группу, она становится способной самостоятельно справляться с проблемами и непредвиденными ситуациями. Члены группы поддерживают друг друга в процессе различных изменений, что помогает им преодолевать трудности.

Хорошо сплоченная команда имеет ряд преимуществ.

1. *Возможность становления стандарта качества группы.* Так как этот стандарт определяется всей группой единогласно, его легче контролировать, чем чужие стандарты, навязываемые группе извне.
2. *Члены команды поддерживают тесные рабочие контакты.* Работая в группе, люди учатся друг у друга. Скованность и затягивание работы, вызванные незнанием или неосведомленностью, уменьшаются по мере того, как происходит взаимное обучение.
3. *Члены команды ознакомлены с деятельностью друг друга.* Этим достигается возможность продолжения работы даже после ухода одного из сотрудников.
4. *Возможно внедрение в практику группы безличного программирования.* Созданная программа должна быть собственностью всей команды, а не отдельной личности.

Безличное программирование [340] означает определенный стиль работы, при котором проекты, программы и документация считаются собственностью всей группы, а не отдельного человека, который занимался их разработкой. Если разработчики воспринимают свою работу именно таким образом, они охотнее отдают ее на проверку другим членам группы, легко воспринимают критику, стремятся работать над усовершенствованием

системы. При этом команда разработчиков становится более сплоченной, так как каждый чувствует себя в ответе за разработку всей системы.

Кроме того, что безличное программирование улучшает качество системной архитектуры, программ и документации, оно также совершенствует взаимоотношения внутри группы, стимулирует непринужденное обсуждение задачи, независимо от социального положения, опыта и пола. В ходе выполнения проекта члены группы более активно поддерживают рабочие отношения между собой. Все эти факторы содействуют сплоченности и тому, что каждый работник воспринимает себя как часть команды.

Многие факторы могут оказать влияние на сплоченность группы, в том числе организационная культура и выражение личностных качеств в группе. Менеджеры могут развивать сплоченность несколькими путями. Можно организовывать социальные мероприятия для работников и их семей. Можно привить группе чувство самобытности, для чего ее надо назвать, определить сущность команды и сферу ее деятельности. Менеджеры должны проводить мероприятия (например, игры и спорт), прямо направленные на создание команды.

Однако по своему опыту могу сказать, что наилучший способ воспитать дух команды — дать возможность каждому почувствовать, что он несет определенную долю ответственности и что ему доверяют, а также гарантировать доступ к проектной информации для всех членов группы. Иногда менеджерам кажется, что они не должны раскрывать определенную информацию. Однако такая линия поведения будет постоянно создавать в группе чувство недоверия. Простой обмен информацией — самый дешевый и эффективный способ дать людям почувствовать себя частью команды.

Однако в сильных и сплоченных группах также могут возникнуть проблемы.

1. *Неосознанное сопротивление смене лидера.* Если лидера сплоченной группы требуется заменить кем-то со стороны, члены группы могут сообща выступить против нового лидера. Работники тратят много времени, сопротивляясь изменениям и новшествам, предлагаемым новым лидером, при этом постепенно падает производительность труда. Поэтому по возможности следует назначать нового лидера из состава группы.
2. *Групповая мысль.* Это название ситуации [189], в которой ценные рабочие качества членов группы разрушаются в результате преданности команде. Выдвижение альтернативных предложений заменяется приверженностью нормам и решениям группы. Любая идея принимается, если большинству это выгодно, альтернативы при этом не рассматриваются.

Чтобы избежать проблемы групповой мысли, нужно организовывать официальные заседания, где члены команды будут вынуждены так или иначе высказывать собственные мысли. Также можно пригласить независимых экспертов для анализа решения группы. Команда должна состоять из людей, которые способны дискуссировать, задавать вопросы и не обращать особого внимания на устоявшееся положение. Во время дискуссий такие люди выступают в качестве яростных спорщиков, постоянно подвергая сомнениям решения группы, побуждая тем самым членов группы обдумывать и оценивать свои действия.

22.2.3. Общение в группе

Для группы по разработке программных продуктов просто необходим развитой коммуникационный фактор, т.е. общение и хорошие средства связи между членами группы. Работники должны информировать друг друга о том, как идет их работа, о решениях, которые предпринимались в отношении проекта, и тех необходимых изменениях, которые вносились в предыдущие распоряжения. Постоянное общение также способствует спло-

ченности, поскольку работники лучше начинают понимать мотивацию своих коллег, их слабые и сильные стороны.

На эффективность общения могут оказать влияние следующие показатели.

1. *Размер группы.* Чем больше группа, тем труднее обеспечить постоянное общение между ее членами. Показатель односторонних связей в группе вычисляется по формуле $n \times (n-1)$, где n — размер группы. Из этого можно понять, что в группе из 7–8 человек могут быть люди с низким показателем связей. Различие в социальном положении членов группы приводит к появлению большего количества односторонних связей. Члены группы с более высоким положением в обществе чаще доминируют в общении со своими стоящими ниже коллегами, которые, в свою очередь, неохотно идут на контакт или высказывают критические соображения.
2. *Структура группы.* Работники, состоящие в группах с неформальной структурой, легче общаются между собой, чем в группах, которые имеют определенную официальную иерархию в отношениях. В последних общение происходит четко в иерархической последовательности (в ту или иную сторону). Сотрудники на одной и той же иерархической ступеньке могут вовсе не общаться между собой. Это в основном проблема широкомасштабных проектов, включающих несколько групп по разработке программных продуктов. Если в таком проекте сотрудники разных отделов общаются только через своих менеджеров, это может привести к запаздыванию в проведении работ и недопониманию друг друга.
3. *Состав группы.* Если в группе много людей с похожими личностными характеристиками, они могут конфликтовать друг с другом, вследствие чего может значительно снизиться уровень общения в группе. Лучше всего люди общаются в смешанных разнополых группах [229], чем в однородных по полу. Среди женщин преобладает внешняя ориентация, поэтому в группе они служат своего рода регуляторами взаимоотношений, облегчающими общение.
4. *Рабочее окружение.* Правильная организация рабочего места — основополагающий фактор в развитии или торможении коммуникационных связей в группе. Этот вопрос рассматривается в разделе 22.3.1.

22.2.4. Организация группы

Небольшие команды программистов лучше всего организовывать в легком неформальном духе. Лидер участвует в работе над программным продуктом наравне с другими членами группы. Техническим лидером может стать человек, который лучше всего будет управлять процессом разработки. Неформальная группа всегда обсуждает предстоящую работу со всеми членами коллектива, а задания назначаются в соответствии с возможностями и опытом конкретного сотрудника. Высокоуровневое проектирование системы осуществляется старшими специалистами, проектирование низкого уровня предоставляется тому сотруднику, который назначен на выполнение конкретного задания.

Работа неформальных групп может оказаться чрезвычайно эффективной, особенно если большинство членов группы опытные и квалифицированные специалисты. Группа функционирует в качестве демократического образования, где решения принимаются большинством голосов. В психологическом плане это порождает дух команды и, следовательно, приводит к укреплению сплоченности и повышению производительности труда. Неформальная организация группы может оказать медвежью услугу в случае, если группа состоит из неопытного или некомпетентного персонала. Возникает нехватка руководяще-

го звена, способного управлять работой, что приводит к недостатку координации работ и, возможно, к провалу проекта.

Бек (Beck, [32]) в своей книге по "экстремальному" программированию описал довольно интересный способ неформальной организации группы. Согласно такому подходу решения, которые входили в круг полномочий менеджера (например, вопросы графика работ), передаются в обязанность коллективу. Программисты работают над кодом в парах и принимают на себя коллективную ответственность за разрабатываемые программы. Согласно результатам отчетов этот механизм подбора группы работает достаточно хорошо. Однако, по моему мнению, данный подход, как и метод "чистая комната" (см. главу 19), для достижения успеха требует участия высококвалифицированного персонала с сильной личностной мотивацией.

Я полагаю, что наиболее сильное влияние на продуктивность труда оказывают личные качества сотрудников (к обсуждению этого вопроса я вернусь в главе 23). Чтобы использовать высококвалифицированный персонал с наибольшей отдачей, многие специалисты [11, 16, 60] предлагают строить группу вокруг одного высококвалифицированного ведущего программиста. Основной принцип такой организации состоит в том, чтобы компетентный и опытный сотрудник отвечал за разработку всего программного продукта. Ведущего программиста не следует загружать рутинной работой, ему наоборот нужна хорошая поддержка в решении вопросов административного и технического плана. Такого сотрудника также следует избавить от излишнего общения со специалистами вне группы (рис. 22.6).

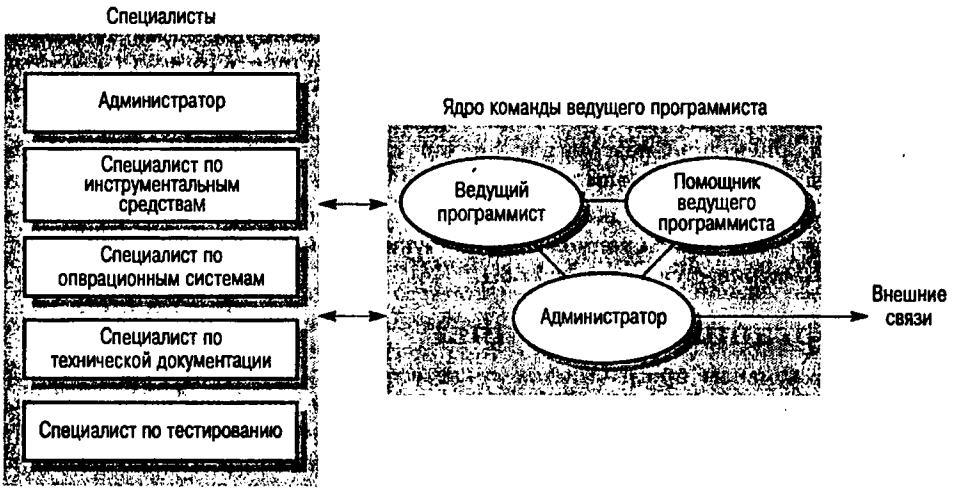


Рис. 22.6. Команда с ведущим программистом

Основными членами (ядром) команды ведущего программиста являются следующие лица.

1. Ведущий программист, который берет на себя основную ответственность за разработку, программирование, тестирование и внедрение системы.
2. Опытный помощник (заместитель) ведущего программиста, чья роль состоит в поддержке ведущего программиста и аттестации программной системы.
3. Администратор, который принимает на себя всю канцелярскую работу, связанную с проектом (например, управление конфигурацией, заключительные процедуры с документацией и т. д.).

В зависимости от типа и размера прикладной задачи, из совокупности специалистов могут быть приглашены профессионалы в качестве временного или постоянного персонала для работы в команде. Это могут быть администратор, специалист по инструментальным средствам разработки ПО, специалист по операционным системам или языкам программирования, специалист по тестированию систем.

Обоснованием такого подхода может служить следующее: сколько специалистов по программному обеспечению, столько различий в способностях программирования. Уровень производительности труда (по условной шкале от наилучших до наихудших программистов) может разниться в 25 раз. Поэтому нужно с наибольшей эффективностью использовать возможности лучших работников, обеспечив им оптимальную поддержку. Хотя идее команды ведущего программиста уже более 25 лет, она все еще остается одним из эффективных способов организации небольших групп программистов.

Если вы можете подобрать нужных людей, то организация группы по такому принципу приведет к успеху. Однако и в таких группах есть свои проблемы.

1. Талантливые разработчики и программисты встречаются нечасто. А организация группы основана на самом компетентном ведущем программисте и его помощнике. Если они совершают ошибки, их решения не с кем обсудить. В демократической группе, наоборот, каждый может обсудить решение и таким образом обнаружить ошибки и избежать их.
2. Ведущий программист ответственен за полное выполнение проекта и может также взять на себя заслуги в случае успеха. Однако члены группы могут с этим не согласиться, если их роль в проекте не будет признана в достаточной мере. В таком случае не удовлетворены их потребности в оценке, так как все заслуги будут приписываться ведущему программисту.
3. Возможность невыполнения проекта в случае болезни или увольнения ведущего программиста и его заместителя. Руководители проектов могут не согласиться на такой риск.
4. Организационная структура компании может оказаться не способной обеспечить подобный тип группы. Большие компании, как правило, имеют хорошо развитую служебную иерархию, поэтому назначение ведущего программиста со стороны может оказаться достаточно трудной задачей. А в компаниях небольшого размера практически невозможно выделить одного сотрудника на выполнение единственного задания.

Поэтому структура групп с ведущим программистом может оказаться для организации весьма рискованной идеей. Однако из нее можно вынести кое-что действительно полезное: необходимо поддерживать талантливых программистов, выделяя для них помощников, администраторов и т.д. Таким образом можно использовать способности одаренных сотрудников наиболее эффективно. Назначение узкоспециализированного специалиста на короткие промежутки времени в отдельные группы разработчиков может стать эффективнее использования программиста с большим опытом на протяжении долгого периода времени в работе над одним проектом.

22.3. Подбор и сохранение персонала

Одной из основных функциональных обязанностей менеджера проекта является подбор персонала. Только в исключительных случаях менеджеры имеют право назначать наиболее подходящих для определенной работы сотрудников, независимо от обязанностей, которые они выполняют в текущий момент, или от финансовых возможностей ком-

пании. В основном менеджер проекта не имеет права окончательного выбора персонала. Он может быть вынужден принимать в команду разработчиков любого сотрудника, который есть в компании и более-менее подходит для этого, причем, как правило, найти подходящего человека необходимо в максимально короткий срок. Также свободному подбору персонала может препятствовать ограниченный бюджет проекта. Именно последний фактор может стать значительным препятствием в найме квалифицированных (но дорогих) программистов, которые могли бы работать над проектом.

В табл. 22.1 показаны основные факторы, которые могут оказать влияние на решение менеджера в выборе будущего персонала, если он имеет для этого относительную свободу. Трудно представить эти факторы в порядке их значимости, поскольку они зависят от области применения разрабатываемого ПО, вида проекта, а также от квалификации и опыта будущих членов группы.

Таблица 22.1. Факторы, влияющие на выбор персонала

Фактор	Пояснение
Знания об области применения ПО	Для того чтобы разработать хорошо функционирующую систему, программист должен иметь четкое представление о той прикладной области, где будет применять разрабатываемое ПО
Опыт работы на многих компьютерных платформах	Этот фактор может оказаться важным при низкоуровневом программировании, в общем случае он не является решающим
Знание языка программирования	Этот фактор применим для краткосрочных проектов, когда просто не хватает времени для изучения нового языка
Образование	Образование служит своеобразным показателем тех основных знаний и умений, которыми должен владеть кандидат, а также его способности к обучению. Этот показатель становится менее значимым пропорционально опыту, получаемому в работе над различными проектами
Коммуникабельность	Этот фактор достаточно важен, так как в процессе реализации проекта программистам нужно будет общаться в устной и в письменной форме с другими специалистами, менеджерами и потребителями
Способность адаптироваться	Показателем может служить разнообразный опыт, полученный ранее. Этот фактор также может показать способность к обучению
Жизненная позиция	Люди, работающие над проектом, должны любить свою работу и стремиться получать новые знания и навыки. Это очень показательный фактор, однако его трудно оценить
Личностные качества	Это очень важный признак, однако также трудный в оценивании. Ведь члены группы должны быть совместимы (в разумной мере, естественно) для общей работы. Не существует отдельного типа личности, который в большей или меньшей степени соответствует специалисту в области инженерии ПО

Решение о назначении нового сотрудника по проекту основывается на трех видах информации.

1. Информация об образовании и практическом опыте, предоставляемая кандидатом на должность (резюме или автобиография).
2. Информация, получаемая при интервьюировании кандидата.
3. Рекомендации от других людей, имеющих опыт совместной работы с кандидатом.

Некоторые компании пользуются разнообразным набором тестов для оценивания кандидатов. Это может быть проверка способности работать программистом и психологические тесты. Такие тесты направлены на создание психологической карты личности, определяют отношение опрашиваемого к работе определенного типа и способность ее выполнить. Некоторые менеджеры убеждены в абсолютной бесполезности таких тестов, другие считают, что с помощью тестов они получают полезную информацию для отбора персонала. Как уже упоминалось ранее, способность решать задачи относится к сфере построения семантических моделей, что само по себе занимает много времени. Тесты на профессиональную пригодность и психологические задачи основаны в большей мере на быстроте ответов на вопросы. До сих пор я не нашел убедительного доказательства определения способности решать практические задачи на основе тестов на профессиональную пригодность.

Если менеджеры проектов сталкиваются с трудностями в подборе подходящего персонала с нужными способностями и опытом, они вынуждены составлять команды из неопытных программистов, что, в свою очередь, вызывает определенные проблемы, связанные с неосведомленностью в области применения ПО или технологий, используемых в проекте. Часто причина такой ситуации кроется в том, что в некоторых организациях технически одаренные сотрудники быстро достигают вершины карьеры. Для дальнейшего продвижения таким сотрудникам необходим статус менеджера. Если переводить их в категорию меиеджеров, для компании это будет означать потерю ценного технически квалифицированного персонала. Чтобы избежать такой ситуации, некоторые компании учредили в своей структуре два параллельных рода деятельности — технический и управленческий, имеющих одинаковую значимость. Опытный технический персонал ценится на таком же уровне, как и менеджеры. С продвижением карьеры специалист может ориентироваться на технический либо управленческий род деятельности и переходить на тот или иной вид деятельности без потери статуса и зарплаты.

22.3.1. Рабочая среда

Организация рабочего места имеет исключительное значение для производительности труда и получения удовлетворения от работы. Психологические исследования доказали влияние на поведение сотрудников размера комнаты, мебели, оборудования, температуры, влажности, яркости и качества света, уровня шума и возможности уединения. Поведение членов группы разработчиков также зависит от архитектурных особенностей помещения и возможностей доступных средств связи.

Вам, как менеджеру, дорого обойдется неумение обеспечить хорошие рабочие условия для сотрудников. Если люди чувствуют себя некомфортно на работе, то пропорционально неудовлетворенности возрастает и коэффициент текучести рабочей силы. Это, в свою очередь, потребует больших затрат средств на наем и обучение нового персонала. Также может замедлиться выполнение проекта, поскольку не хватает квалифицированных специалистов [93].

Команды по разработке программных продуктов часто работают в больших открытых офисах, иногда с перегородками, и только старшим по должности сотрудникам могут предоставляться отдельные кабинеты. В статье [93] приведены результаты исследования, которые показали, что открытые помещения, используемые многими фирмами, абсолютно

непопулярны и столь же непродуктивны. В этом исследовании определены ключевые факторы оформления рабочего пространства.

1. *Уединенность.* Программистам требуется определенное место, где они могут сконцентрироваться и работать без чьего-либо вмешательства.
2. *Обзор внешнего мира.* Люди предпочитают работать при дневном свете, имея перед собой привлекательный вид из окна.
3. *Индивидуализация.* Люди усваивают различные привычки в организации рабочего процесса и имеют разные мнения относительно оформления помещения. Здесь большое значение имеет способность устраивать рабочее место так, чтобы оно отвечало разным вкусам и несло индивидуальные черты.

Таким образом, человек по своей природе предпочитает работать в отдельном помещении, которое он может оформить по своему вкусу и желанию. Такие кабинеты, в отличие от открытых офисов, создают более благоприятную обстановку и снижают количество пауз в работе. В открытом помещении сотруднику невозможно сконцентрироваться, как это можно сделать в тихой уединенной рабочей обстановке. Следствием снижения концентрации становится непременное падение уровня производительности труда.

Следовательно, отдельные помещения для программистов оказывают существенное влияние на производительность. В [92] проведен сравнительный анализ продуктивности труда программистов в различных рабочих условиях. В результате выяснилось, что возможность сконцентрироваться и снижение уровня посторонних вмешательств в работу значительно повышает рабочую активность. Из двух групп программистов с одинаковыми способностями работающие в благоприятных условиях оказались в два раза продуктивнее, чем те, которые находились в худших помещениях.

Группы по разработке программного обеспечения также нуждаются в помещении, где все члены группы могут собраться вместе и обсудить проект (будь то официальное заседание или просто неформальная встреча). Комната для встреч должны вмещать всех членов группы и обеспечивать им необходимую степень уединенности. Два вида потребностей, уединение и общение внутри группы, могут показаться взаимоисключающими. Мак-Кью (McCue, [233]) предлагает решить подобный конфликт потребностей путем размещения индивидуальных кабинетов вокруг одной большой комнаты для встреч (рис. 22.7).

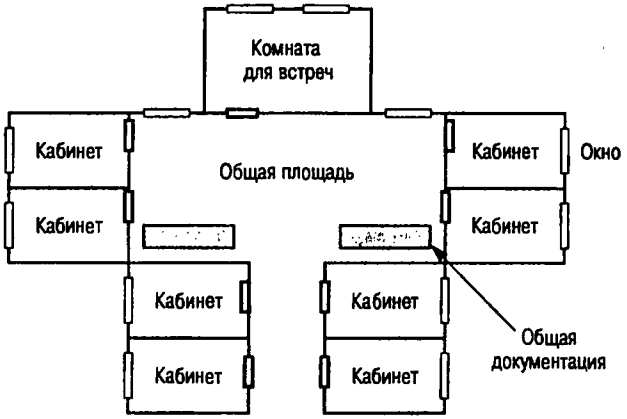


Рис. 22.7. Размещение кабинетов и комнаты для встреч

Похожую модель организации офисов предлагает Бек (Beck) при описании "экстремального" программирования [32]. Однако он все же настаивает на сохранении открытого типа офисов для совместной работы с перегородками для тех сотрудников, которые хотели бы поработать в одиночестве. Ключевым фактором здесь является предоставление двух видов помещения (общих и индивидуальных) с тем, чтобы члены группы выбирали рабочее место по своему усмотрению.

Такой тип организации рабочего помещения помогает сотрудникам решать проблемные вопросы и обмениваться информацией неформальным, но эффективным способом. В статье [340] приводится довольно забавный случай, когда организация пыталась бороться с таким явлением, как "трата времени" сотрудников за болтовней вокруг кофеварки. Руководство распорядилось забрать у них кофеварку, после чего немедленно стали поступать многочисленные запросы на оказание помощи в программировании. Оказывается, в то время, когда сотрудники сплетничали за чашкой кофе, они заодно помогали друг другу в решении проблемных вопросов. Этот случай неоспоримо доказывает, что комнаты для неформальных встреч так же необходимы организации, как и комнаты для рабочих заседаний.

22.4. Модель оценки уровня развития персонала

Институт инженерии программного обеспечения (Software Engineering Institute – SEI) в США долгое время занимался программой усовершенствования процесса создания ПО. Модель CMM (Capability Maturity Model – модель оценки уровня развития) является частью этой программы. Описание этой модели вы найдете в главе 25. Она вобрала в себя наилучшее из практики инженерии программного обеспечения. В развитие этой модели институт также предлагает модель оценки уровня развития персонала (People Capability Maturity Model – P-CMM) [83]. Ее можно использовать в качестве основы стратегии управления человеческими ресурсами в организации.

Подобно CMM, модель P-CMM имеет пять уровней (рис. 22.8).

1. *Начальный уровень.* Практикуется управление персоналом, не оформленное в виде определенных правил.
2. *Повторение.* Проведение политики, направленной на развитие способностей персонала.
3. *Становление.* Введение в организации стандарта управления, основанного на лучшем опыте управления персоналом.
4. *Управление.* Определяются и вводятся количественные цели в управлении персоналом.
5. *Оптимизация.* Центр внимания переносится на непрерывное повышение квалификации и мотивации работников.

В книге [83] определяются стратегические цели этой модели.

1. Расширение возможностей организаций, занимающихся программным обеспечением, путем повышения квалификации персонала.
2. Гарантирование того, что способность высококачественной разработки программного обеспечения является отличительной чертой всей организации, а не тех нескольких людей, которые этим занимаются.

3. Обеспечение совместимости между мотивацией отдельного индивидуума и мотивацией всей организации.
4. Сохранение в организации ценных человеческих ресурсов (например, сотрудников, имеющих редкие знания и навыки).

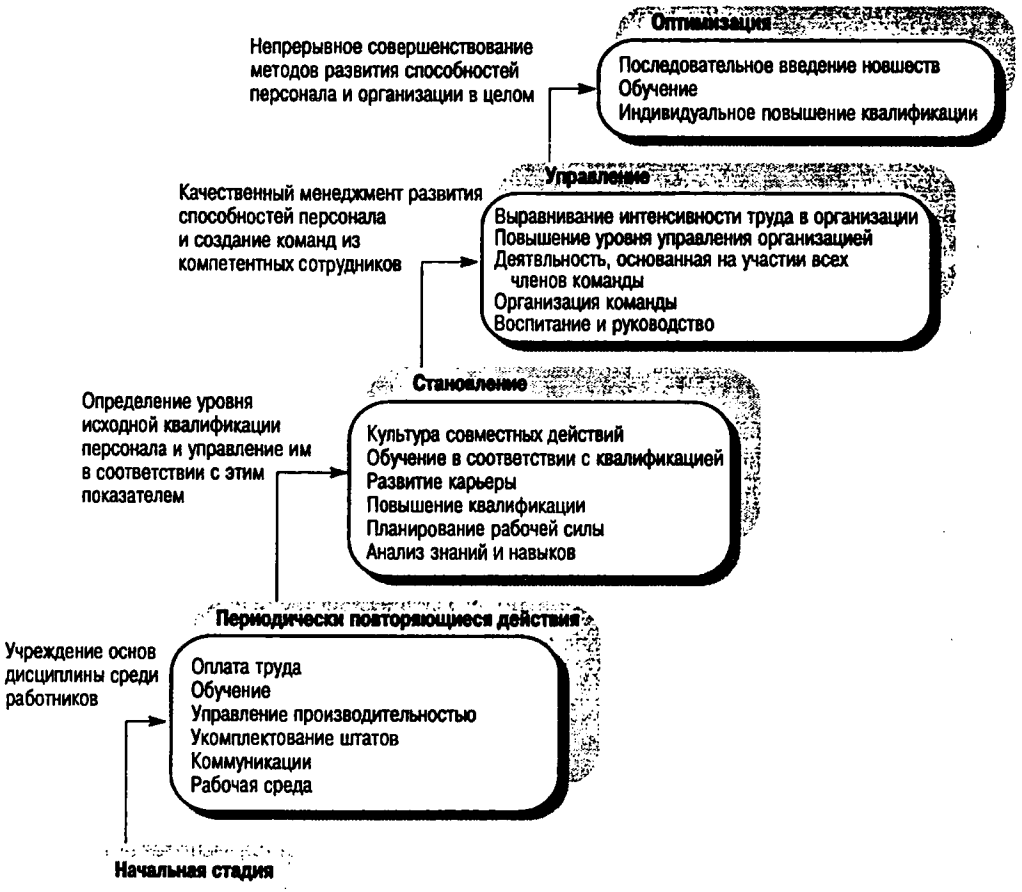


Рис. 22.8. Модель оценки уровня развития персонала

Модель P-CMM – это отличный и действенный подход к улучшению качества управления персоналом, так как он дает основы для мотивирования, признания, нормализации и совершенствования лучшего опыта в управлении персоналом. Благодаря этому мы осознаем значимость работника как личности и необходимость его дальнейшего совершенствования. Для внедрения данной модели в полном объеме потребуется значительное количество средств, а многим организациям это просто не нужно. Однако эта модель в любом случае является отличным пособием, которое поможет многим организациям усовершенствовать свои возможности в производстве программного обеспечения высокого качества.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Управление разработкой программного обеспечения напрямую связано с управлением людьми. Поэтому менеджер программного проекта должен иметь определенное представление о психологии и субъективных факторах человеческой деятельности, чтобы не ставить невыполнимых требований для себя и для персонала.
- Человеческая память подразделяется на кратковременную, промежуточную и долговременную. Знания могут быть синтаксическими и семантическими. Процесс решения проблемы включает в себя интеграцию семантической информации, хранящейся в долговременной памяти, и новой информации, поступающей из кратковременной памяти.
- Знание предметной области, в которой будет применяться создаваемое ПО, способность адаптироваться и выражение личностных качеств являются ключевыми факторами при подборе персонала.
- Группы по разработке ПО должны быть небольшими и хорошо сплоченными. Лидеры таких групп должны иметь высокую техническую квалификацию и обеспечиваться административной и технической поддержкой.
- На взаимоотношения внутри группы влияют многие факторы, в том числе такие, как социальное положение членов группы, размер группы, распределение персонала по половому признаку, личностные качества и возможность общения.
- Путем обеспечения надлежащих условий работы, которые включают наличие соответствующих вычислительных и коммуникационных средств, можно повысить производительность труда и чувство удовлетворенности работой.
- Модель оценки уровня развития персонала P-CMM обеспечивает общую основу и механизм повышения производительности труда персонала, а также возможность получать наилучший результат от инвестиций в человеческие ресурсы.

Упражнения

22.1. Дайте краткое описание иерархической структуры человеческой памяти. Объясните, почему данной структурой обеспечивается лучшее понимание объектно-ориентированных систем, чем систем, построенных на функциональной декомпозиции.

22.2. Каково различие между синтаксическими и семантическими знаниями? Исходя из вашего личного опыта, дайте несколько примеров этих двух типов знания.

22.3. Представьте себе, что как менеджер по подготовке персонала вы несете ответственность за обучение основам языка программирования выпускников университетов, которые будут работать в вашей компании над разработкой систем противовоздушной обороны. В основном в разработке используется язык программирования Ada, который специально был создан для программирования систем противовоздушной обороны. Стажеры могут иметь степень бакалавра в программировании, инженерной или физической науках. У некоторых может быть опыт программирования, но никто раньше не сталкивался с языком Ada. Расскажите, как вы планируете организовать процесс обучения для данной группы стажеров.

22.4. Какие факторы прежде всего принимаются во внимание при подборе сотрудников для работы над программным проектом?

22.5. Объясните, каким образом доступность информации о ходе разработки проекта и тех технических решениях, которые имеют отношение ко всем членам группы, могут усилить сплоченность группы.

22.6. Дайте определение понятия "групповая мысль". Опишите, какие затруднения могут возникать в результате этого явления и как их можно избежать.

22.7. Представьте, что вы менеджер и вас попросили спасти проект, от которого зависит финансовый успех или неуспех всей компании. Руководство старшего уровня передало вам незакрытый бюджет, и вам предстоит набрать команду из пяти человек, работающих в данный момент над другими проектами компании. Кроме того, конкурирующая фирма, специализирующаяся в этой же области, активно набирает персонал, и некоторые ваши сотрудники перешли к конкурентам.

Опишите две модели организации команды программистов, которые применимы в данной ситуации, и выберите одну из них. Дайте обоснование своему выбору и объясните причину, по которой вы отказались от альтернативной модели.

22.8. Почему открытые и общие помещения менее пригодны для работы команды программистов, чем индивидуальные кабинеты? В каких случаях, по вашему мнению, открытые офисы оказываются более подходящими?

22.9. Почему модель P-CMM считается эффективной основой для повышения качества управления персоналом? Дайте предложение по видоизменению данной модели с тем, чтобы приспособить ее к применению в небольших компаниях. Должны ли менеджеры вести себя дружелюбно и стирать социальные рамки в общении с нижестоящими членами группы?

22.10. Как вы думаете, порядочно ли схитрить и дать те ответы на вопросы в психологическом тесте, которые работодатель хочет от вас услышать, а не говорить того, что вы на самом деле думаете?

23 Оценка стоимости программного продукта

Цели

Цель этой главы — представить на рассмотрение различные методы оценки стоимости и затрат, необходимых для создания программного продукта. Прочитав ее, вы должны:

- знать, как оценивается себестоимость и назначается цена программного продукта, и понимать сложные взаимосвязи между этими двумя процессами;
- знать три системы оценивания производства программного продукта;
- понимать, что для правильной оценки стоимости ПО и для создания графика работ необходимо применять широкий спектр различных методов оценки стоимости программного продукта;
- знать принципы работы модели COCOMO 2 для алгоритмической оценки стоимости.

Содержание

В главе 4 был описан процесс планирования программного проекта. Под этим подразумевается разбивка проекта на ряд этапов, выполняемых параллельно или последовательно. Ранее процесс планирования рассматривался в аспекте определения этих этапов, их взаимозависимости и распределения специалистов по работам, выполняемым на каждом этапе.

В этой главе я вернусь к общей проблеме оценки затрат и времени, необходимых для выполнения определенных этапов проекта. Менеджеры проекта должны научиться давать правильные ответы на следующие вопросы.

1. Какие затраты необходимы для выполнения этапа?
2. Сколько это займет времени?
3. Какова стоимость выполнения данного этапа?

Оценка стоимости проекта и планирование графика работ проводятся параллельно. Однако некоторые предварительные расчеты должны быть выполнены на ранней стадии, еще до начала разработки точного плана проекта. Такие расчеты необходимы для утверждения бюджета проекта или для выставления цены заказчику.

Как только проект начинает действовать, все расчеты должны регулярно обновляться. Это помогает планировать работу и содействует эффективному использованию средств. Если фактические расходы значительно превышают планируемые, менеджеру необходимо предпринять какие-либо действия. Это может быть перечисление дополнительных средств на проект либо изменение будущих этапов работ в соответствии с фактическим бюджетом.

Обычно для оценки проекта по разработке программного обеспечения используются три параметра.

- Стоимость аппаратных средств и программного обеспечения, включая их обслуживание.
- Расходы на командировки и обучение.
- Расходы на персонал, в основном на привлечение со стороны специалистов по программному обеспечению.

В большинстве проектов доминируют расходы на персонал. Компьютеры, имеющие достаточно мощности для разработки программного продукта, в наше время относительно дешевые. Значительными могут быть затраты на командировки, если проект разрабатывается в разных местах, однако для большинства проектов они все же не очень существенны. Более того, расходы на командировки можно сократить, используя вместо них электронную почту, факс или телеконференции.

Расходы на персонал — это не только оплата труда работников. В них могут включаться накладные расходы, т.е. все расходы, которые касаются работы организации, деленные на количество работающего персонала. Таким образом, общая сумма расходов на персонал состоит из нескольких статей расходов.

1. Расходы на содержание, отопление и освещение офисов.
2. Расходы на содержание вспомогательного персонала — бухгалтеров, секретарей, уборщиц и технического персонала.
3. Расходы на содержание компьютерной сети и средств связи.
4. Расходы на централизованные услуги — библиотеки, места отдыха и развлечения и т.д.

5. Расходы на социальное обеспечение и выплаты служащим (например, пенсии и медицинская страховка).

Обычно накладные расходы приравниваются к удвоенной зарплате программиста, в зависимости от размера компании и расходов на ее содержание. Например, если специалист по программному обеспечению получает 90 000 долларов в год, расходы организации на этот год составляют сумму 180 000 долларов, или 15 000 долларов в месяц.

Оценка стоимости должна быть объективной, чтобы дать компании-разработчику достаточно точный прогноз себестоимости проекта. Если себестоимость рассчитывается для включения в коммерческое предложение заказчику, следует принять решение о том, какую цену назначить за проект. Традиционно в цену продукта включают издержки производства плюс предлагаемая прибыль. Однако определить соотношение себестоимости проекта и цены, выставляемой заказчику, не всегда просто.

На определение цены программного продукта могут повлиять организационные, экономические, политические и коммерческие соображения. Эти факторы показаны в табл. 23.1. Таким образом, взаимоотношения цены и себестоимости совсем не так просты, как может показаться. Кроме того, необходимо учитывать глобальные цели организации, поэтому назначением цены за программный продукт вместе с менеджерами проектов занимается и старший руководящий состав компании.

Таблица 23.1. Факторы, влияющие на стоимость программного продукта

Фактор	Описание
Возможности рынка ПО	Организация-разработчик может выставить низкие цены на программный продукт из-за намерения переместиться в другой сегмент рынка ПО. Даже если организация примет более низкую прибыль в первом проекте, это все равно может привести к более высоким доходам в будущем, поскольку полученный опыт позволит заниматься разработкой подобных программных продуктов и в дальнейшем
Невозможно учесть все факторы, влияющие на стоимость	Если организация примет фиксированную величину стоимости, издержки производства могут возрасти из-за непредвиденных расходов
Условия контракта	Заказчик может позволить разработчику сохранить за собой право владения программным кодом с последующим его использованием в других проектах. При этом назначенная цена может быть ниже, чем в том случае, если право на программный код передано заказчику
Изменение требований	При изменении требований к ПО организация может снизить цену с тем, чтобы выиграть контракт. Если контракт уже заключен, за изменение требований можно назначить дополнительную цену
Финансовая стабильность	Фирмы, испытывающие финансовые затруднения, для получения заказа могут снизить цены на свои разработки. Как правило, лучше сегодня получить более низкую прибыль или даже работать на уровне самоокупаемости, чем обанкротиться в будущем

23.1. Производительность

Производительность в промышленности обычно измеряется путем деления количества единиц выпущенной продукции на количество человеко-часов, необходимых для их производства. Однако в области разработки По любая задача имеет несколько вариантов решения, каждый со своими особенностями. Одно проектное решение отличается эффективным способом выполнения, другое имеет программный код, который легко читается либо удобен в эксплуатации. Поэтому не имеет смысла сравнивать уровни производительности при разработке решений с разными качественными характеристиками.

Несмотря на это, менеджеры могут оценить производительность самих специалистов по разработке программного обеспечения. Это понадобится при оценивании проекта и определеиии эффективности усовершенствования процесса и технологии разработки ПО. Оценка производительности в этом случае будет основана на измеренин количественных показателей программных продуктов и последующем делении их на количество усилий, затраченных на разработку этих продуктов. При этом можно использовать два типа показателей.

1. *Показатель размера.* Зависит от размера выходного результата очередного этапа работ. Наиболее часто применяемым критерием такого типа является количество строк разработанного программного кода. За аналогичный показатель также можно взять количество инструкций объектной программы или количество страниц системной документации.
2. *Функциональный показатель.* Зависит от функциональных возможностей программного продукта в целом. Производительность в этом случае выражается количеством полезных выполняемых функций, разработанных в определенный отрезок времени. К наиболее распространенным показателям этого типа относится количество функциональных и объектных точек.

Количество строк программного кода за человеко-месяц — наиболее популярный критерий оценки производительности. Он определяется путем деления общего количества строк кода на количество времени в человеко-месяцах, которое потребуется для завершения проекта. Это время, потраченное на анализ, проектирование, кодирование, тестирование и разработку документации программного продукта.

Данный подход впервые появился еще во время массового использования таких языков программирования, как FORTRAN, язык ассемблера и COBOL. Затем программы переводились на перфокарты, каждая из которых содержала по одному оператору. Таким образом, было легко подсчитывать количество строк кода. Оно соответствовало количеству перфокарт в колоде. Однако программы, написанные на языках типа Java или C++, состоят из описаний, выполняемых операторов и комментариев. Они также могут включать макрокоманды, которые состоят из нескольких строк кода. С другой стороны, в одной строке может находиться не один, а несколько операторов. Таким образом, соотношения между операторами и строками в листинге могут быть достаточно сложными.

Одни методы подсчета строк основываются только на выполняемых операторах; другие подсчитывают выполняемые операторы и объявления данных; третьи ведут подсчет всех строк программы, независимо от их содержимого. Были предложены определенные стандарты подсчета строк в различных языках программирования [269], однако они не приобрели широкой популярности. Поэтому практически невозможно сравнивать производительность различных компаний-разработчиков, если только все они не используют один и тот же метод подсчета строк кода.

Также может ввести в заблуждение и сравнение производительности программистов, пишущих на разных языках программирования. Чем выразительнее язык, тем ниже производительность. Это "странное" утверждение объясняется тем, что при оценивании производительности создания ПО берется во внимание вся деятельность по разработке программного продукта, тогда как данная система измерения основывается лишь на оценивании процесса программирования.

Для примера возьмем систему, которая должна быть написана с помощью кода ассемблера (5000 строк) или с помощью языка высокого уровня (1500 строк). Время выполнения различных этапов создания ПО показано в табл. 23.2. Специалист, программирующий на языке ассемблера, будет иметь производительность 714 строк в месяц, а у программиста, работающего с языком высокого уровня, производительность будет в два раза ниже — 300 строк в месяц. И это несмотря на то, что программирование на языке высокого уровня стоит дешевле и занимает меньше времени.

Таблица 23.2. Время выполнения этапов разработки программной системы

	Анализ (недели)	Проектирование (недели)	Кодирование (иедели)	Тестирование (недели)	Документирование (недели)
Язык ассемблера	3	5	8	10	2
Язык высокого уровня	3	5	8	6	2

	Размер (строки кода)	Затраченное время (недели)	Производительность (строк в месяц)
Язык ассемблера	5000	28	714
Язык высокого уровня	1500	20	300

Альтернативой показателю размера при оценивании производительности может служить функциональный показатель. В этом случае можно избежать тех "аномалий" в оценке производительности, которые встречаются при использовании показателя размера, так как функциональность не зависит от языка программирования. В статье [228] дается краткое описание и сравнение способов оценки производительности, основанных на функциональных показателях.

Одним из наиболее распространенных методов этого типа является метод функциональных точек. Впервые он был предложен в работе [6], а затем его усовершенствованный вариант был представлен в статье [7]. Функциональные точки не зависят от применяемого языка программирования, благодаря чему появилась возможность сравнения производительности разработки программных систем, написанных на различных языках программирования. Критерием оценки производительности выступает количество функциональных точек, создаиных за человеко-месяц. Функциональная точка — это не какая-то отдельная характеристика программного продукта, а целая комбинация его свойств. Под-

счет общего количества функциональных точек в программе проводится путем измерения или оценивания следующих свойств программы.

- Интенсивность использования ввода и вывода внешних данных.
- Взаимодействие системы с пользователем.
- Внешние интерфейсы.
- Файлы, используемые системой.

Сложность каждого из указанных критериев оценивается отдельно, в результате каждому критерию присваивается определенная весовая величина, которая может колебаться от 3 (для простого ввода данных) до 15 баллов за использование сложных внутренних файлов. Можно использовать весовые величины, предложенные в статье [7], или величины, основанные на личном опыте.

Нескорректированный подсчет функциональных точек (unadjusted function-point count — UFC) выполняется путем вычисления суммы произведений оценки каждого фактора (количество элементов, составляющих данный фактор) на выбранную весовую величину этого фактора:

UFC = Σ (количество элементов данного типа) × (весовая величина).

Первоначальный метод подсчета функциональных точек был в дальнейшем усовершенствован путем добавления тех факторов, значение которых зависит от общей сложности проекта. Здесь принимается во внимание степень распределенности обработки данных, многократность использования программных элементов, качество функционирования и т.п. Значение, полученное при нескорректированном подсчете функциональных точек, нужно умножить на факторы, определяющие сложность проекта, в результате будет получено итоговое значение.

Вместе с тем в [330] отмечено, что оценка сложности несет в себе также и субъективный фактор, так как подсчет функциональных точек зависит от лица, проводящего оценивание. Люди имеют разные понятия о сложности. Так как подсчет функциональных точек зависит от мнения оценивающего, существует множество вариаций подсчета функциональных точек. Это приводит к разным взглядам на значимость функциональных точек [122]. Однако многие заявляют, что, несмотря на все недостатки, на практике этот метод себя оправдывает [196].

Альтернативой функциональным точкам являются объектные точки [19], особенно если при разработке ПО используется язык программирования четвертого поколения. Метод объектных точек применяется в модели оценивания COCOMO 2, которая рассматривается далее в главе. (Объектные точки — это отнюдь не классы объектов, которые производятся в результате применения объектно-ориентированного подхода в работе над программой, что можно было бы предположить, исходя из названия.) Количество объектных точек в программе можно получить путем предварительного подсчета ряда элементов.

1. Количество изображений на дисплее. Простые изображения принимаются за 1 объектную точку, изображения умеренной сложности принимаются за 2 точки, а очень сложные изображения принято считать за 3 точки.
2. Количество представленных отчетов. Для простых отчетов назначаются 2 объектные точки, умеренно сложным отчетам назначаются 5 точек. Написание сложных отчетов оценивается в 8 точек.
3. Количество модулей, которые написаны на языках третьего поколения и разработаны в дополнение к коду, написанному на языке программирования четвертого поколения. Каждый модуль на языке третьего поколения считается за 10 объектных точек.

Преимущество данного метода состоит в том, что объектные точки легко оценить исходя из высокоуровневой спецификации программного продукта, поскольку они связаны с конкретными объектами – изображениями, отчетами и модулями на языках программирования третьего поколения.

Количество функциональных и объектных точек можно оценить уже на ранней стадии выполнения проекта. Оценку этих параметров можно начинать сразу после разработки внешних взаимосвязей системы. Именно на этой стадии очень сложно провести точную оценку размера программы, беря за основу только строки кода. Более того, язык программирования на этом этапе может быть еще не выбран. Вместе с тем оценка на ранней стадии особенно необходима, если используются модели алгоритмического оценивания себестоимости, которые рассматриваются далее.

Подсчет функциональных точек можно проводить параллельно с методом подсчета количества строк кода. Количество функциональных точек при этом используется для оценивания окончательной величины кода. На основе анализа выполнения предыдущих программных проектов для определенных языков программирования можно оценить среднее количество строк кода (average number of lines of code – AVC), необходимых для реализации одной функциональной точки. В этом случае получим оценку размер кода нового проекта, рассчитанную следующим образом:

размер кода = AVC × количество функциональных точек.

Значение AVC варьируется от 200 до 300 строк кода на одну функциональную точку в языке ассемблера и от 2 до 40 строк кода для языков программирования четвертого поколения.

Производительность отдельных программистов, работающих в организации-разработчике, зависит от множества факторов, влияющих на их работу. Некоторые из наиболее значимых факторов представлены в табл. 23.3. Однако различия в индивидуальных способностях программистов намного важнее всех этих факторов. В исследовании ранних этапов программирования [305] отмечено, что производительность некоторых программистов в 10 раз выше, чем у других. Это наблюдение подтверждается и моим личным опытом. Большие команды, члены которых имеют необходимый набор личностных качеств и способностей, будут иметь "среднюю" производительность. Вместе с тем в командах небольшого размера общая производительность во многом зависит от индивидуальных умений и навыков каждого члена команды.

Должен отметить, что не существует какой-либо величины, определяющей "среднюю" производительность программиста, которую можно было бы применять в разных организациях и при создании разных программных продуктов. Например, при разработке больших систем производительность может быть очень низкой и доходить до 30 строк за человеко-месяц. На простых программных системах производительность может подняться до 900 строк в месяц. В [44] высказано предположение, что производительность программиста может колебаться от 4 до 50 объектных точек в месяц, в зависимости от наличия средств поддержки и от способностей программиста.

Недостатком оценок, которые основываются на подсчете объема выполненной работы или определении количества затраченного времени, является то, что они не принимают во внимание такие важные нефункциональные свойства разрабатываемой системы, как надежность, удобство эксплуатации и т.п. Обычно здесь работает простое правило: больше – значит, лучше. Бек (Beck, [32]) приводит удачное замечание, что если вы работаете над постоянным усовершенствованием и упрощением системы, то подсчет строк ничего не даст.

Таблица 23.3. Факторы, влияющие на производительность программиста

Фактор	Описание
Опыт разработки ПО для данной предметной области	Для эффективной разработки программного продукта необходимо знание той предметной области, где будет эксплуатироваться разрабатываемое ПО. Инженеры, имеющие понятие об этой предметной области, выявят наивысшую производительность
Процесс управления качеством	Применяемый метод программирования может оказать существенное влияние на производительность написания кода. Этот фактор рассматривается в главе 25
Размер проекта	Чем больше проект, тем больше времени уходит на согласование различных вопросов внутри группы разработчиков. Тем самым уменьшается время, расходуемое непосредственно на разработку ПО, и снижается производительность
Поддержка технологии разработки ПО	Хорошая поддержка технологии разработки ПО, например CASE-средства или системы управления конфигурацией, может значительно повысить производительность труда программиста
Рабочая обстановка	Как уже упоминалось в главе 22, спокойное рабочее окружение с индивидуальными рабочими местами способствует повышению производительности

Описанные методы также не учитывают многократность использования программного продукта. В действительности необходимо оценить стоимость повторного использования определенной системы с данным набором функциональных и качественных характеристик, имеющей собственные показатели удобства сопровождения и т.д. Все эти параметры только косвенно соотносятся с такими количественными показателями, как, например, размер системы.

Трудности также могут возникнуть в случае, если менеджеры используют показатели производительности для оценивания способностей персонала. В этом случае качество выполненной программистом работы может отойти на второй план по отношению к производительности. Может случиться, что "менее продуктивный" программист создаст код, который будет надежнее, понятнее и дешевле в использовании. Поэтому нельзя пользоваться показателями производительности как единственным источником оценивания труда программиста.

23.2. Методы оценивания

Не существует простого метода определения будущих затрат, необходимых для разработки программного продукта. Начальное оценивание можно провести, основываясь на определенных пользовательских требованиях высокого уровня. Но заранее не известно, какие технологии будут применяться при разработке ПО и какие компьютеры будет использоваться. Также невозможно предугадать, какие люди будут работать над проектом и какие у них будут навыки и опыт. Это показывает, что чрезвычайно трудно провести точную оценку стоимости проекта на самом раннем этапе. Более того, основная проблема в оценке себестоимости проектов заключается в низкой точности применяемых методов оценивания.

Часто в расчет себестоимости проекта закладывается его окупаемость. Таким образом, первоначальная оценка себестоимости определяет бюджет проекта, за рамки которого нельзя выходить при реализации проекта. Вместе с тем я не знаю ни одного реализованного проекта, где бы стоимость не корректировалась по ходу его выполнения. Но всегда в ходе реализации проекта фактические расходы сравниваются с предварительной оценкой затрат.

Несмотря ни на что, организации-разработчики обязательно должны оценивать затраты на разработку и себестоимость программного продукта. Для этого можно применять методы, описанные в табл. 23.4 [46].

Таблица 23.4. Методы оценки себестоимости

Метод	Описание
Алгоритмическое моделирование себестоимости	Метод основан на анализе статистических данных о ранее выполненных проектах, при этом определяется зависимость себестоимости проекта от какого-нибудь количественного показателя программного продукта (обычно это размер программного кода). Проводится оценка этого показателя для данного проекта, после чего с помощью модели прогнозируются будущие затраты
Оценка эксперта	Проводится опрос нескольких экспертов по технологии разработки ПО, знающих область применения создаваемого программного продукта. Каждый из них дает свою оценку себестоимости проекта. Потом все оценки сравниваются и обсуждаются. Этот процесс повторяется до тех пор, пока не будет достигнуто согласие по окончательному варианту предварительной сметы проекта
Оценка по аналогии	Этот метод используется в том случае, если в данной области применения создаваемого ПО уже реализованы аналогичные проекты. В таком случае при оценке затрат для сравнения берутся предыдущие проекты. В книге [251] дано достаточно ясное описание этого метода
Закон Паркинсона	Согласно этому закону усилия, затраченные на работу, распределяются равномерно по выделенному на проект времени. Здесь критерием для оценки затрат по проекту являются человеческие ресурсы, а не целевая оценка самого программного продукта. Если проект, над которым работает пять человек, должен быть закончен в течение 12 месяцев, то затраты на его выполнение исчисляются в 60 человеко-месяцев
Назначение цены с целью выиграть контракт	Затраты на проект определяются наличием тех средств, которые имеются у заказчика. Поэтому себестоимость проекта зависит от бюджета заказчика, а не от функциональных характеристик создаваемого продукта

В интересной статье [161] описан эксперимент, в котором менеджеров попросили провести предварительную оценку размера будущей программной системы и необходимых для ее разработки затрат. Менеджеры при этом использовали мнения экспертов и оценку по аналогии. Результаты эксперимента показали, что менеджеры провели достаточно точную оценку затрат, однако определение размера будущей системы при этом было менее точным. Это означает, что оценка затрат, основанная только на данных о размере программы, будет неточной.

Методы предварительной оценки себестоимости могут выполняться с применением нисходящего или восходящего подходов. При нисходящем подходе оценка себестоимости начинается на уровне системы: рассматриваются функциональные возможности программы в целом и то, как эти возможности реализуются посредством функций более низкого уровня. Здесь учитывается себестоимость таких этапов разработки, как сборка системы, управление конфигурацией и создание технической документации.

В отличие от нисходящего подхода, восходящий начинается на уровне системных компонентов. Система разбивается на компоненты и определяются затраты на разработку каждого из них. Затем эти затраты суммируются для определения полной стоимости проекта.

Недостатки восходящего подхода являются достоинствами нисходящего и наоборот. Восходящий подход может недооценить затраты, необходимые для решения сложных проблем, возникающих при разработке таких специфических компонентов системы, как интерфейсы для нестандартных аппаратных средств. Детального обоснования для составления сметы затрат в этом случае не существует. Нисходящий подход, напротив, дает такое обоснование, а также возможность рассмотреть каждый компонент в отдельности. Вместе с тем данный подход больше акцентирует внимание на таких этапах реализации проекта, как, например, сборка системы. Кроме того, нисходящий подход является более дорогостоящим. Для его применения нужно иметь хотя бы предварительный результат проектирования системы с тем, чтобы оценить каждый ее компонент.

Каждый метод оценивания, безусловно, имеет слабые и сильные стороны. Для работы с большими проектами необходимо применить несколько методов оценивания себестоимости для их последующего сравнения. Если при этом получаются совершенно разные результаты, значит, информации для получения более точной оценки недостаточно. В этом случае необходимо воспользоваться дополнительной информацией, после чего повторить оценивание, и так до тех пор, пока результаты разных методов не станут достаточно близкими.

Описанные методы оценивания применимы, если документированы требования для будущей системы. В таком случае существует возможность определить функциональные характеристики разрабатываемой системы. Обычно все большие проекты разработки ПО имеют документ, в котором определены требования к системе.

Однако во многих проектах оценка затрат проводится только на основании проекта требований к системе. В этом случае лица, участвующие в оценке стоимости проекта, будут иметь минимум информации для работы. Процедуры анализа требований и создания спецификации весьма дорогостоящи. Поэтому менеджерам компании следует составить смету на их выполнение еще до утверждения бюджета для всего проекта.

Во многих случаях популярной становится стратегия ценообразования с целью "выиграть контракт". Можно согласиться, что данная фраза звучит несколько некорректно и не по-деловому, однако эта стратегия на самом деле себя оправдывает. Стоимость работы согласовывается на основании предварительного проекта предложения. Далее проводятся переговоры между компанией-исполнителем и заказчиком с тем, чтобы обсудить детальное техническое задание, которое, однако, ограничивается согласованной суммой. Продавец и заказчик также должны обсудить приемлемые функциональные возможности системы. Здесь основополагающим фактором для многих проектов становятся возможности бюджета, а не требования к системе. Требования всегда можно изменить так, чтобы не выходить за рамки принятого бюджета.

При оценке себестоимости проекта менеджеры должны всегда помнить о том, что между прошлыми проектами и будущими разработками может быть существенная разница. Только за последние 10 лет на свет появился целый ряд новейших разработок и технологий. Многие менеджеры очень мало ознакомлены, а иногда просто не имеют понятия об этих технологиях и о том, какое влияние они могут оказать на проект. Вот несколько примеров технических и технологических новшеств, которые могут повлиять на оценку стоимости проекта, основанную на предыдущем опыте.

- Появление объектно-ориентированного программирования вместо процедурного.
- Применение систем типа клиент/сервер вместо систем, основанных на мэйнфреймах.
- Применение готовых коммерческих пакетов программного обеспечения вместо собственной разработки компонентов системы.
- Повторное использование компонентов системы вместо новых разработок.

- Использование CASE-средств и генераторов программ вместо разработки ПО без применения средств поддержки.

Все эти факторы отнюдь не облегчают задачу менеджера в оценке стоимости программной продукции. И в этом случае предыдущий опыт не всегда оказывается полезным для проведения такой оценки.

23.3. Алгоритмическое моделирование стоимости

Алгоритмическое моделирование считается наиболее системным подходом к определению стоимости, однако это не значит, что он всегда дает точные результаты. Алгоритмическую модель стоимости можно построить с помощью анализа затрат и параметров уже разработанных проектов. Для прогнозирования затрат применяется математическая формула, в которой учтены данные о размере проекта, количестве программистов, а также другие факторы и процессы. В работе [198] приведены 13 алгоритмических моделей прогнозирования затрат, построенных на анализе выполнения предыдущих проектов.

В большинстве алгоритмических моделей формулы вычисления затрат имеют экспоненциальный вид. Причина этого – отсутствие линейной зависимости себестоимости проекта от его размера. С увеличением проекта появляются дополнительные расходы, связанные с ростом затрат на коммуникации, усложнением управления конфигурацией, увеличением объема работ по сборке системы и т.д. Оценки затрат также могут умножаться на коэффициенты, учитывающие свойства разрабатываемого программного продукта, платформу разработки, технологию создания ПО и квалификацию привлеченных специалистов.

В общем случае формула для вычисления алгоритмической оценки стоимости записывается следующим образом:

$$\text{затраты} = A \times \text{размер}^{B} \times M,$$

где A – постоянный коэффициент, который зависит от организации выполнения проекта и типа разрабатываемого программного обеспечения; показатель размер может соотноситься либо с размером кода программы, либо с функциональной оценкой, выраженной в количестве объектных или функциональных точек; показатель степени B может варьироваться в пределах от 1 до 1.5, он отображает объем работ, требующийся для реализации больших проектов; множитель M отображает характеристики различных этапов разработки, а также характеристики создаваемого продукта.

Алгоритмическим моделям присущи общие проблемы.

1. На ранней стадии выполнения проекта бывает сложно определить показатель размер при наличии информации только о системных требованиях. Несмотря на то что оценка, основанная на функциональных, или объектных, точках, проще оценки размера кода, результаты тоже не всегда будут точными.
2. Оценка факторов, которые влияют на показатели B и M, носит субъективный характер. Значения этих показателей могут отличаться, если этим занимаются люди с разным опытом и квалификацией.

Основой для многих алгоритмических моделей оценки себестоимости является количество строк программного кода в созданной системе. Оценку размера кода можно получить по аналогии с другими проектами путем преобразования функциональных точек в размер кода, сравнения размеров компонентов системы и сравнения с эталонными компонентами, а также просто на основе инженерной интуиции.

На размер окончательной системы могут повлиять решения, которые были приняты в процессе реализации проекта уже после утверждения начальной сметы. Например, это может быть решение о том, использовать ли в создаваемом ПО, требующем сложной системы управления

данными, базы данных сторонних производителей или разработать свои системы управления данными. Очевидно, что при использовании сторонних баз данных программный код, который следует создать, будет меньшего размера. Кроме того, имеет значение и язык программирования. Для таких языков, как Java, потребуется больше строк кода, чем если бы применялся, скажем, язык C. В то же время "лишний" код на языке Java потребует провести больше проверок программы в процессе компиляции, вследствие чего расходы на аттестацию системы наверняка снизятся. Чему в данном случае отдать предпочтение? Кроме прочего, также необходимо оценить объем повторного использования кода.

При использовании алгоритмических моделей для ценообразования проекта в них должен учитываться тип проекта, при этом результаты оценивания необходимо тщательно анализировать. Менеджер должен составить не одну, а несколько оценок стоимости (среди них наименее выгодную, ожидаемую и наиболее выгодную). Следует также помнить о большой вероятности значительных ошибок при раннем прогнозировании себестоимости. Наиболее точные оценки можно получить в том случае, если создаваемый продукт хорошо структурирован, модель учитывает интересы организации-заказчика, заранее определены язык программирования и необходимые аппаратные средства.

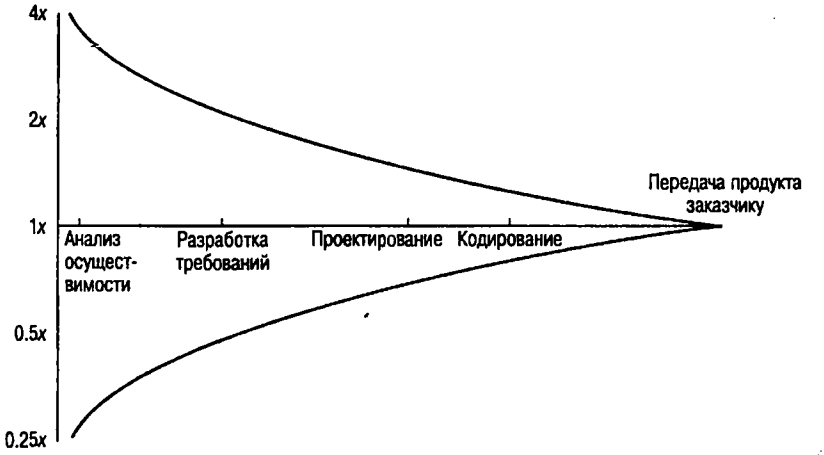


Рис. 23.1. Изменчивость оценивания затрат

Точность результатов прогнозирования себестоимости также зависит от количества информации о создаваемой системе. По ходу реализации проекта увеличивается количество информации, вследствие чего оценка себестоимости становится все более точной. Если при начальном оценивании временных затрат для разработки системы требовалось x месяцев, то реальная длительность выполнения проекта может колебаться от $0.25x$ до $4x$. Однако этот диапазон постепенно сужается в процессе выполнения проекта, как показано на рис. 23.1. Эта схема была заимствована из статьи [44] и основана на опыте реализации многих проектов разработки программных продуктов.

23.3.1. Модель COCOMO

Существует целый ряд алгоритмических моделей для прогнозирования затрат и себестоимости, а также создания графика работ для программных проектов. Принципиально они между собой не отличаются, хотя используют значения разных параметров. Одной из самых интересных моделей я считаю COCOMO[1]. Эта модель основана на опыте реализации многих программных проектов. Она создана путем сбора данных о большом коли-

[1] *COCOMO (от COnstructive COst MOdel) – конструктивная стоимостная модель.* – Прим. ред.

честве проектов и анализа этой информации, в результате чего получены формулы, наилучшим образом аппроксимирующие имеющиеся данные. Я отдаю предпочтение модели COCOMO по трем причинам.

1. Эта модель имеет хорошую техническую документацию, общедоступна, существуют коммерческие программные средства ее поддержки.
2. Модель популярна и ценится среди широкого круга пользователей.
3. Она прошла достаточно долгий путь развития со времени первого появления в 1981 году [46], была усовершенствована для разработки ПО на языке Ada [43], последняя версия модели опубликована в 1995 году [44].

Модель COCOMO в первом варианте (известном сейчас как COCOMO 81) имела трехуровневую структуру, где уровни определяли сложность анализа себестоимости. На первом (или базовом) уровне проводилась начальная грубая оценка, на втором уровне эта оценка уточнялась путем применения различных множителей, учитывающих особенности проекта и технологии разработки ПО, самый сложный уровень дает возможность рассчитать себестоимость для разных стадий проекта. В табл. 23.5 показаны основные формулы модели COCOMO для проектов с различной степенью сложности. Здесь множитель M идентичен тому, который будет описан далее для COCOMO 2.

Моделью COCOMO 81 предусмотрена разработка программного обеспечения в соответствии с каскадной моделью, причем предполагается, что бо́льшая часть системы разрабатывается "с нуля". Одиако со времени первой версии данной модели было сделано несколько фундаментальных изменений в целя́х ее усовершенствования. Теперь модель допускает производство ПО путем компоновки повторно используемых компонентов, связывая их между собой с помощью какого-либо языка сценариев. Сегодня прототипирование и пошаговая разработка — наиболее распространенные модели создания программного продукта. Во многих случаях используются серийные компоненты, доступные на рынке программных продуктов. Кроме того, существующие программы модифицируются в целях создания нового программного обеспечения. Поддержка CASE-средств уже стала доступной для многих видов работ по созданию ПО.

Таблица 23.5. Модель COCOMO 81

Сложность проекта	Формула[2]	Описание
Простой проект	$PM = 2.4\ (KDSI)^{1.05} \times M$	Простые проекты для небольших команд разработчиков
Средней сложности	$PM = 3.0\ (KDSI)^{1.12} \times M$	Более сложные проекты, при разработке которых члены команды могут ощущать нехватку опыта и знаний соответствующих систем
Проект встроенной системы	$PM = 3.6\ (KDSI)^{1.20} \times M$	Сложные проекты, где ПО является частью комплекса аппаратных и программных средств, других технических механизмов и устройств

Принимая во внимание все эти изменения, модель COCOMO 2 допускает самые разнообразные подходы к процессу разработки программных продуктов: прототипирование, сборку систем из отдельных компонентов, использование языков программирования четвертого поколения и т.д. Но теперь уровни модели не только отображают возрастающую сложность определения себестоимости разработки ПО, но и учитывают этапы работы над

[2] *В этих формулах PM (от Person-Months) обозначает человеко-месяцы, KDSI (от thousand (Kilo-) of Delivered Source Instructions) – количество инструкций (в тысячах) в конечной программе (общепринятая единица измерения объема работ по программированию).* – Прим. ред.

программой, что позволяет провести предварительную оценку себестоимости на ранних этапах выполнения проекта с последующей ее детализацией после определения архитектуры системы. Модель COCOMO 2 охватывает три описанных ниже уровня.

1. *Уровень предварительного прототипирования.* Для определения необходимых затрат осуществляется оценка размера системы на основе объектных точек прототипа с помощью простой формулы "размер–производительность".
2. *Уровень предварительного проектирования.* Этот уровень предусматривает окончание работы над системными требованиями и, возможно, над начальным проектом архитектуры программы. Оценка затрат на этом уровне основана на функциональных точках, которые затем пересчитываются в количество строк кода программ. Здесь используются формулы, подобные описанным выше, с соответствующим набором множителей.
3. *Постархитектурный уровень.* После разработки архитектуры системы существует реальная возможность достаточно точно оценить размер программы. Однако оценка на этом уровне уже будет включать более расширенный ассортимент множителей, которые должны отражать возможности персонала, а также характеристики создаваемого программного продукта и проекта в целом.

Уровень предварительного прототипирования

Этот уровень был введен в модель COCOMO для оценки затрат на прототипирование проектов, а также для тех проектов, в которых программное обеспечение разрабатывалось путем сборки уже существующих компонентов. Здесь оценка затрат основана на подсчете взвешенных объектных точек (это понятие рассматривается в разделе 23.1), деленных на стандартное значение производительности. Производительность зависит от опыта и способностей разработчика, а также от возможностей CASE-средств, используемых для поддержки процесса разработки. В табл. 23.6. показаны различные уровни производительности, которые были предложены разработчиками модели [44].

Таблица 23.6. Производительность, выраженная в объектных точках

Опыт и возможности программиста	Очень низкие	Низкие	Средние	Высокие	Очень высокие
Уровень и возможности CASE-средств	Очень низкие	Низкие	Средние	Высокие	Очень высокие
Производительность (количество объектных точек в месяц)	4	7	13	25	50

На этом уровне повторное использование компонентов не редкость, поэтому прогнозируемое количество объектных точек должно учитывать процентное соотношение (долю) повторного использования компонентов (далее обозначено как % многократного использования). Таким образом, формула для предварительного определения объема работ будет выглядеть следующим образом:

PM = (NOP × (1 – %многократного использования/100))/ PROD.

Здесь PM – это затраты, выраженные в человеко-месяцах, NOP – количество объектных точек, а PROD – производительность, как показано в табл. 23.6.

Уровень предварительного проектирования

На этом этапе оценка основана на стандартной формуле алгоритмических моделей, а именно:

$$\text{затраты} = A \times \text{размер}^B \times M.$$

Основываясь на собственном банке данных, Боэм (Boehm) предложил значение коэффициента A принять равным 2.5, если оценка осуществляется на этом уровне. Размер системы выражается в KSLOC, что означает количество строк программного кода в тысячах. Он определяется путем подсчета количества функциональных точек в программе и перевода их в KSLOC с помощью стандартных таблиц, которые определяют соотношение размера программы с функциональными точками для различных языков программирования. Эта оценка размера применима скорее к кодам, написанным вручную, нежели к генерированным или повторно используемым.

Показатель степени B отражает затраты, которые увеличиваются по мере увеличения размера проекта. Это не постоянная величина, как в предыдущей версии модели COCOMO, она изменяется от 1.1 до 1.24, что зависит от того, насколько новаторским является данный проект, от гибкости процесса разработки ПО, от применяемых процессов управления рисками, сплоченности команды программистов и уровня управления организацией-разработчиком (см. главу 25). О том, как рассчитывается этот показатель, речь идет в следующем разделе.

Множитель M является произведением семи показателей, характеризующих проект и процесс создания ПО, а именно: надежность и уровень сложности разрабатываемой системы (RCPX), повторное использование компонентов (RUSE), сложность платформы разработки (PDIF), возможности персонала (PERS), опыт персонала (PREX), график работ (SCED) и средства поддержки (FCIL). Это позволяет провести оценивание по шестибалльной шкале, где число 1 будет соответствовать самым малым значениям этих показателей, а число 6 — самым высоким значениям. С другой стороны, их можно вычислить путем комбинирования значений более детализированных показателей, которые используются на постархитектурном уровне.

Таким образом, оценка затрат вычисляется по следующей формуле:

$$PM = A \times \text{размер}^B \times M + PM_m,$$

где $M = PERS \times RCPX \times RUSE \times PDIF \times PREX \times FCIL \times SCED$.

Последнее слагаемое в формуле (PM_m) обозначает фактор, используемый в случаях, когда значительная часть кода генерируется автоматически. При этом часть кода всегда требуется вводить вручную, но уровень производительности все равно будет выше, чем при полностью ручном вводе. Затраты PM_m рассчитываются отдельно по приведенной ниже формуле, а затем добавляются к оценке затрат на введенный вручную код.

$$PM_m = (ASLOC \times (AT/100))/ ATPROD,$$

где ASLOC — это количество строк кода, произведенных автоматическим способом, ATPROD — уровень производительности автоматической генерации кода. Однако следует заметить, что требуется также выполнение определенных работ для согласования сгенерированного кода с остальной частью системы. Объем этих работ зависит от процента автоматически сгенерированного кода во всей системе (коэффициент AT). Фактически производительность зависит от количества созданных программных модулей. Чем меньше объем сгенерированного кода, тем больший объем работ необходим для интеграции его с другими кодами системы.

Постархитектурный уровень

Для получения оценки на постархитектурном уровне используется такая же формула, как и для оценки на уровне предварительного проектирования. На этом уровне оценка будет более точной, для предварительной оценки затрат будут использоваться уже не семь, а семнадцать показателей.

Здесь при оценке количества строк программного кода учитываются два важных фактора.

- *Возможность изменения системных требований.* Следует учитывать повторные работы, которые необходимо выполнить вследствие изменения системных требований. Оценка этих работ выражается в количестве строк программного кода, которое необходимо изменить, и затем прибавляется к предварительной оценке размера системы.
- *Степень повторного использования компонентов.* Если степень повторного использования программных компонентов значительна, в оценку количества строк разрабатываемого кода необходимо внести поправки. Однако в [309] показано, что расходы на повторное использование компонентов не всегда линейно зависят от размера этих компонентов, так как требуют затрат на их подбор и на то, чтобы разобраться с их возможностями и интерфейсами; кроме того, могут потребоваться затраты на внесение изменений в эти компоненты.

Оценка затрат на повторное использование компонентов в модели COCOMO 2 рассчитывается по следующей формуле:

$$ESLOC = ASLOC \times (AA + SU + 0.4\,DM + 0.3\,CM + 0.3\,IM)/100.$$

Здесь ESLOC — количество строк нового кода, ASLOC — количество строк повторно используемого кода, требующего изменений, DM — процент изменений в архитектуре системы, CM — процент измененного кода, а IM — процент затрат на интеграцию повторно используемого программного обеспечения. Коэффициент SU зависит от затрат на адаптацию повторно используемых программных компонентов и колеблется в пределах от 50 (для сложного неструктурированного кода) до 10 (для грамотно написанного объектно-ориентированного кода). Коэффициент AA отображает затраты на начальную оценку возможности повторного использования компонентов. Его значение колеблется от 0 до 8.

Показатель степени в формуле расчета затрат в модели COCOMO 1 имеет три возможных значения, которые соотносятся с различными уровнями сложности проекта. С возрастанием уровня сложности проекта увеличивается значимость размера системы. Однако отрицательный эффект размера системы можно нивелировать с помощью организационных мероприятий, что учтено в модели COCOMO 2. Здесь показатель степени рассчитывается с учетом пяти показателей, которые описаны в табл. 23.7. Они отсчитываются по шестибалльной шкале от низшего (5 баллов) до наивысшего (0 баллов) уровня. Значения показателей суммируются, сумма делится на 100, результат прибавляется к числу 1.01, после чего получается значение показателя степени.

Таблица 23.7. Показатели, используемые при расчете показателя степени в модели COCOMO 2

Показатель	Пояснение
Новизна проекта	Отображает предыдущий опыт организации в реализации проектов данного типа. Очень низкий уровень этого показателя означает отсутствие опыта, наивысший уровень указывает на компетентность организации-разработчика в данной области ПО

Окончание табл. 23.7

Показатель	Пояснение
Гибкость процесса разработки	Отображает возможность изменения процесса разработки ПО. Очень низкий уровень этого показателя означает, что процесс определен заказчиком заранее, наивысший – заказчик определил лишь общие задачи без указания конкретной технологии процесса разработки ПО
Анализ архитектуры системы и рисков	Отображает степень детализации анализа рисков, основанного на анализе архитектуры системы. Очень низкий уровень данного показателя соответствует поверхностному анализу рисков, наивысший уровень означает, что был проведен тщательный и полный анализ всевозможных рисков
Сплоченность команды	Отображает степень сплоченности команды и их способность работать совместно. Очень низкий уровень этого показателя означает, что взаимоотношения в команде сложные, а наивысший – что команда сплоченная и эффективная в работе, не имеет проблем во взаимоотношениях
Уровень развития процесса разработки	Отображает уровень развития процесса создания ПО в организации-разработчике. Оценка этого показателя основывается на вопроснике модели CMM (см. главу 25)

Приведем пример. Предположим, организация-разработчик выполняет программный проект в той области, в которой у нее мало опыта разработок. Заказчик ПО не определил технологический процесс, который будет использовать при создании программного продукта, а также не выделил в плане работ времени на анализ возможных рисков. Для создания программной системы необходимо сформировать новую команду специалистов. Организация-разработчик недавно привела в действие программу усовершенствования технологического процесса разработки ПО и может котироваться как организация второго уровня в соответствии с моделью оценки уровня развития CMM (об этой модели речь идет в главе 25). Для оценки показателя степени используются перечисленные ниже показатели проекта.

1. *Новизна проекта.* Это новый проект для организации, данный показатель имеет низкий уровень (оценивается в 4 балла).
2. *Гибкость процесса разработки.* Нет вмешательства заказчика – уровень показателя очень высокий (оценивается в 1 балл).
3. *Анализ архитектуры системы и рисков.* Анализ не был проведен – уровень данного показателя очень низкий (оценивается в 5 баллов).
4. *Сплоченность команды.* Команда разработчиков новая, информация о ней отсутствует – уровень этого показателя оценивается как обычный (3 балла).
5. *Уровень развития процесса разработки.* Определенное управление проектом имеет место – показатель оценивается как обычный (3 балла).

Сумма значений всех этих показателей составляет 16 баллов, поэтому значение показателя степени будет равно 1.17.

Проектные характеристики, используемые для уточнения предварительной оценки затрат на постархитектурном уровне (табл. 23.8), разбиваются на четыре группы.

1. Характеристики программного продукта, которые определяются системными требованиями.
2. Характеристики аппаратных средств, представляющие собой ограничения, накладываемые на разрабатываемое ПО выбранной платформой вычислительных средств.
3. Характеристики персонала, которые учитывают опыт и возможности специалистов, работающих над проектом.
4. Характеристики проекта, учитывающие определенные параметры и показатели проекта разработки ПО.

Таблица 23.8. Проектные характеристики, формирующие стоимость проекта

Характеристики программного продукта	
RELY	Требуемая надежность системы
CPLX	Сложность системных модулей
DOCU	Объем необходимой документации
DATA	Размер используемой базы данных
RUSE	Процент повторного использования компонентов
Характеристики аппаратных средств	
TIME	Показатели, ограничивающие время исполнения
PVOL	Возможность изменения платформы разработки
STOR	Ограничение объема памяти
Характеристики персонала	
ACAP	Способности лиц, выполняющих анализ проекта
PCON	Сплоченность команды разработчиков
PEXP	Опыт программирования в данной области ПО
PCAP	Способности программистов
AEXP	Опыт аналитика проекта в данной области ПО
LTEX	Опыт применения данного языка программирования и средств разработки
Характеристики проекта	
TOOL	Использование вспомогательных программных средств
SCED	Уплотнение графика работ
SITE	Количество работ, выполняемых в разных местах, и качество коммуникаций

В табл. 23.9 показан пример того, каким образом эти характеристики влияют на предварительную оценку затрат. Я взял показатель степени 1.17, полученный в предыдущем примере, и предположил, что RELY, CPLX, STOR, TOOL и SCED являются основными характеристиками, формирующими стоимость проекта. Все другие характеристики имеют значение 1, поэтому не влияют на оценку затрат.

Таблица 23.9. Расчет оценки затрат

Значение показателя степени	1.17
Размер системы (с учетом повторного использования компонентов и возможного изменения требований)	128 000 DSI[3]
Начальная оценка по модели COCOMO без учета проектных характеристик	**730 человеко-месяцев**
Надежность системы	Очень высокая, множитель 1.39
Сложность системных модулей	Очень высокая, множитель 1.3
Ограничение объема памяти	Высокое, множитель 1.21
Использование вспомогательных средств	Низкое, множитель 1.12
График работ	Ускоренный, множитель 1.29
Уточненная оценка по модели COCOMO	**2306 человеко-месяцев**
Надежность системы	Очень низкая, множитель 0.75
Сложность системных модулей	Очень низкая, множитель 0.75
Ограничение объема памяти	Нет, множитель 1
Использование вспомогательных средств	Очень высокое, множитель 0.72
График работ	Нормальный, множитель 1
Уточненная оценка по модели COCOMO	**295 человеко-месяцев**

В этом примере я присвоил максимальные и минимальные значения ключевым характеристикам с тем, чтобы показать, каким образом они влияют на оценку затрат. Значения были взяты из руководства по модели COCOMO 2 [41]. Вы можете сами убедиться, что высокие значения для характеристик, влияющих на формирование стоимости, привели к увеличению оценки затрат более чем в три раза, тогда как при низких значениях оценка затрат была снижена почти в три раза по сравнению с начальной. Этот пример показывает отличия разных типов проектов, а также трудности переноса опыта разработок из одной области ПО в другую.

Хотя формула, предложенная разработчиками модели COCOMO, отображает их опыт разработчиков и основана на большой базе данных, однако я полагаю, что эта модель излишне сложна для практического использования. Слишком много показателей необходимо учитывать и слишком широкие рамки для определения их значений. Таким образом, каждый, кто хочет воспользоваться этой моделью, должен выверить и приспособить ее к своим данным, накопленным при реализации предыдущих проектов, так как именно они дадут информацию о тех частных обстоятельствах, которые могут оказать влияние на ход выполнения данного проекта. Некоторые организации накопили достаточно информации о прошлых проектах в той форме, которая способна проверить и настроить модель COCOMO.

23.3.2. Алгоритмические модели стоимости в планировании проекта

Алгоритмические модели стоимости применяются для сравнения различных инвестиций в целях снижения стоимости проекта. Это особенно важно в тех случаях, когда при-

[3] *DSI (Delivered Source Instructions) – количество инструкций в конечной программе.* – Прим. ред.

нимаются решения в отношении покупки аппаратных или программных средств либо возникает необходимость найма новых сотрудников с особыми навыками, нужными для реализации проекта.

В качестве примера рассмотрим встроенную систему для управления экспериментом, который будет проводиться в космосе. Оборудование для проводимых в космосе экспериментов должно быть предельно надежным и подлежит строгим весовым ограничениям. Поэтому, например, может появиться необходимость свести к минимума количество чипов на монтажной плате. Если взять для оценки модель COCOMO, то множители, зависящие от ограничений в проекте и надежности, будут иметь значение больше единицы.

Стоимость проекта складывается из трех компонентов.

1. Стоимость целевых аппаратных средств, на которых будет функционировать разрабатываемая система.
2. Стоимость платформы (вычислительная техника плюс программное обеспечение), используемой для разработки системы.
3. Стоимость затрат на разработку системы.

На рис. 23.2 представлены некоторые проектные показатели и характеристики, которые могут учитываться при расчете стоимости. Например, снижение стоимости ПО может потребовать бо́льших затрат на целевые аппаратные средства или вложения дополнительных средств в усовершенствованные инструменты разработки.

Дополнительные расходы на аппаратные средства в данном случае допустимы, так как разрабатываемая система является узкоспециализированной и не предназначена для массовой продажи. Если же аппаратные средства встраивается в коммерческие продукты, то дополнительные вложения в целевые аппаратные средства (для снижения стоимости ПО) используются редко, потому что это повышает цену продаваемого продукта.

Рис. 23.2. Проектные показатели и характеристики, учитываемые при расчете стоимости

В табл. 23.10 показаны аппаратные и программные средства, обеспечивающие реализацию характеристик A–E, описанных на рис. 23.2. Стоимость данного проекта в соответствии с моделью COCOMO 2 (без учета проектных характеристик (множителей), влияющих на формирование стоимости, см. предыдущий раздел) оценивается в 45 человеко-месяцев. Стоимость затрат на один человеко-месяц составляет $15 000.

Соответствующие множители, влияющие на формирование стоимости, учитывают ограниченность времени исполнения и объема памяти (показатели TIME и STORE), доступность средств поддержки системы разработки, таких, как кроссплатформенные компиляторы и т.п. (показатель TOOL), и опыт команды разработчиков (показатель LTEX). Для всех проектных характеристик множитель RELY (показатель надежности) равен 1.39; это означает, что для разработки надежной системы требуются дополнительные затраты.

Стоимость программного продукта (SC) вычисляется следующим образом:

SC = оценка затрат × RELY × TIME × STOR × TOOL × EXP × $15000.

Проекту с характеристикой A соответствует стоимость разработки системы с уже имеющимся персоналом и средствами поддержки. Проекты с другими характеристиками требуют расходов либо на аппаратные средства, либо на наем нового персонала (с соответствующими расходами и степенью риска). На примере проекта с характеристикой Б видно, что модернизация аппаратных средств не обязательно снизит затраты, так как множитель, учитывающий опыт команды разработчиков, в этом случае будет очень весомым. Также видно, что модернизация только памяти окажется более эффективной, чем усовершенствование всей вычислительной системы.

Таблица 23.10. Параметры стоимости

Характеристика	RELY	STOR	TIME	TOOL	LTEX	Общие затраты	Стоимость ПО	Стоимость аппаратных средств	Итоговая стоимость
А	1.39	1.06	1.11	0.86	1	63	949 393	100 000	1 049 393
Б	1.39	1	1	1.12	1.22	88	1 313 550	120 000	1 402 025
В	1.39	1	1.11	0.86	1	60	895 653	105 000	1 000 653
Г	1.39	1.06	1.11	0.86	0.84	51	769 008	100 000	897 490
Д	1.39	1	1	0.72	1.22	56	844 425	220 000	1 044 159
Е	1.39	1	1	1.12	0.84	57	851 180	120 000	1 002 706

Проектная характеристика Г обеспечивает самые низкие затраты на разработку системы. Здесь нет необходимости в дополнительных затратах на аппаратные средства, зато нужно нанимать новый персонал для работы по проекту. Если такой персонал уже имеется в наличии в организации-разработчике, тогда этот вариант будет наиболее выгодным и выбор следует остановить именно на нем. Если же нет, то наем нового персонала порождает дополнительные расходы и риски. Это может привести к тому, что преимущества в стоимости могут оказаться не такими значительными, как показано в табл. 23.10. В проекте с характеристикой В сэкономлено почти $50 000, при этом риск практически отсутствует. Менеджеры проектов с консервативным складом ума предпочтут этот вариант более рискованному варианту Г.

Приведенный анализ показал важность такого показателя, как опыт персонала. Если организация наймет квалифицированных сотрудников с большим опытом, это может значительно снизить стоимость проекта. Данный показатель напрямую связан с факторами производительности, описанными в разделе 23.1. Этот пример также показывает, что

вложения в новые аппаратные средства могут оказаться не слишком выгодными. Обычно такую стратегию предпочитают программисты, которые любят работать с новыми системами. Однако недостаток опыта работы с новыми системами в данном случае может имеет более сильное отрицательное влияние на стоимость проекта, чем те преимущества, которые ожидаются от приобретения новых аппаратных средств.

23.4. Продолжительность проекта и наем персонала

Кроме расчета затрат, необходимых для работы над проектом, и определения стоимости этих затрат, менеджеры проектов также должны определить длительность выполнения проекта (т.е. составить временной график работ) и время начала найма персонала для непосредственной работы. Чаще всего компании-разработчики "сжимают" график работ до минимума с тем, чтобы доставить программный продукт на рынок раньше своих конкурентов.

Взаимосвязь между количеством работающих над проектом сотрудников, общими затратами и длительностью разработки не является прямолинейной. С увеличением количества сотрудников возрастают затраты, в том числе временные, поскольку больше времени будет уходить на общение внутри группы разработчиков. Также больше времени потребуется для определения интерфейсов между частями системы, которые они будут разрабатывать. Увеличивая вдвое количество персонала, нельзя гарантировать, что время разработки сократится также вдвое.

Модель COCOMO включает формулу для определения календарного времени (TDEV) реализации проекта. Для всех уровней модели COCOMO существует единая формула расчета времени:

$$TDEV = 3 \times (PM)^{(0.33 + 0.2*(B - 1.01))},$$

где PM – оценка затрат в человеко-месяцах, B – показатель степени, способ определения которого описан ранее (B равен единице для уровня предварительного прототипирования). С помощью этой формулы определяется прогнозируемая длительность проекта.

Однако прогнозируемая длительность проекта и продолжительность графика работ, который определяется планом выполнения проекта, – это не одно и то же. Планируемый график работ может оказаться длиннее или короче прогнозируемой длительности проекта. Разность между этими двумя длительностями учитывается в модели COCOMO 2:

$$TDEV = 3 \times (PM)^{(0.33 + 0.2*(B - 1.01))} \times \%SCED/100,$$

где %SCED – процент увеличения (или уменьшения) прогнозируемой длительности проекта. Значительные расхождения прогнозируемой длительности с планируемым графиком работ означают неминуемые проблемы в процессе реализации проекта.

Рассмотрим пример вычисления длительности проекта по модели COCOMO в предположении, что по предварительной оценке затрат для реализации проекта требуется 60 человеко-месяцев (проект с характеристикой B из табл. 23.10.). Примем число 1.17 за значение показателя степени B. Тогда

$$TDEV = 3(60)^{0.36} = 13 \text{ месяцев.}$$

Если в данном случае нет расхождения с длительностью графика работ, последнее выражение в формуле длительности не влияет на окончательный результат.

Интересным в модели COCOMO может показаться то, что время, требующееся для реализации проекта, — это функция всех затрат по проекту. Причем оно не зависит от количества программистов, работающих над проектом. Это подтверждает идею о том, что не имеет смысла включать большее количество программистов в отстающий от графика

проект с тем, чтобы ускорить его завершение. В статье [251] исследован вопрос ускорения реализации проекта и сделан вывод, что в работе неизбежны проблемы, если для разработки ПО ие выделяется достаточно времени.

Распределяя прогнозируемые затраты на реализацию проекта, мы все же не можем точно знать, сколько человек необходимо включить в команду разработчиков. Часто набор программистов происходит по принципу от меньшего к большему с последующим постепенным уменьшением их численности.

Это объясняется тем, что на раннем этапе разработки требуется относительно небольшое количество специалистов, которые будут заниматься только планнрованием системы и разработкой спецификации. По мере выполнения проекта увеличивается и объем выполняемых работ, а количество персонала увеличивается до максимума. После кодирования и тестирования системы количество специалистов уменьшается, пока не остается один-два человека. Очень быстрое наращивание количества персонала часто указывает на отставание проекта от графика работ. Менеджерам не следует набирать слишком много специалистов на самом раннем этапе выполнения проекта.

Наращивание объема работ можно смоделировать с помощью так называемой кривой Рэлея [223]; оценочная модель Путмана (Putnam, [286]) использует модель наращивания персонала, основанную на этих кривых. Модель Путмана учитывает время разработки как ключевой фактор, поскольку с уменьшением времени на разработку экспоненциально увеличиваются затраты на создание системы.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Основными факторами, влияющими на производительность, являются индивидуальные способности работников (доминирующий фактор), опыт создания ПО данного типа, технология процесса разработки, размер проекта, наличие средств поддержки и рабочая обстановка.
- Существует множество методов оценки стоимости программного продукта, которые следует использовать параллельно. Если полученные результаты имеют большие отличия, значит, для анализа использована недостаточная или неподходящая информация.
- Цена программного продукта часто определяется с целью заключения контракта, что приводит к последующей подгонке функциональных возможностей системы для приведения ее в соответствие с этой ценой.
- Основной трудностью в алгоритмическом моделировании стоимости является зависимость оценки стоимости от свойств и параметров готового продукта. На ранней стадии проекта невозможно точное определение этих свойств и параметров.
- Модель оценки стоимости COCOMO — хорошо продуманная модель, в которой учитываются проектные характеристики, свойства создаваемого ПО, аппаратные средства и возможности персонала. В данную модель также включены средства для определения длительности работ над проектом.
- Алгоритмические модели определения стоимости используются для управления программными проектами, поскольку они поддерживают количественный анализ параметров. Эти модели позволяют определить вклад каждого отдельного параметра в общую стоимость проекта и провести объективное (хотя и не застрахованное от ошибок) сравнение этих параметров.
- Время выполнения работ не зависит прямо пропорционально от количества нанятых для проекта специалистов. Привлечение большего количества людей к проекту, который отстает от графика работ, может еще больше затянуть время его выполнения.

Упражнения

23.1. Опишите два подхода к определению производительности программиста. Отметьте преимущества и недостатки каждого подхода.

23.2. Приведите пять факторов, которые оказывают существенное влияние на производительность команды программистов по разработке больших встроенных систем реального времени.

23.3. Любой оценке стоимости присущ определенный риск, независимо от метода оценки. Предложите четыре способа снижения возможного риска при оценке стоимости.

23.4. Менеджер проекта создания ПО ответственен за разработку системы управления оборудованием по лучевой терапии для лечения людей с онкологическими заболеваниями. Эта система встроена в аппаратные средства и должна работать на специальном процессоре с ограниченным объемом памяти (8 Мбайт). Машина осуществляет связь с базой данных пациента и по окончании лечения автоматически записывает в нее дозу радиации, которой подвергался пациент, а также другие подробности лечения.

23.5. Для определения затрат, необходимых для разработки данной системы, использована модель COCOMO. В результате получена оценка объема работ в 26 человеко-месяцев. При оценивании все множители, формирующие стоимость, были приравнены к единице.

23.6. Объясните, почему эту оценку необходимо уточнить с учетом таких показателей, как проектные характеристики, возможности персонала, параметры системы и организационный фактор. Выделите четыре показателя, которые окажут существенное влияние на первоначальную оценку стоимости по модели COCOMO, а также предложите возможные значения для этих показателей. Обоснуйте выбор каждого показателя.

23.7. Назовите три причины, по которым алгоритмические оценки стоимости, проведенные различными компаниями, не будут сопоставимыми.

23.8. Объясните, каким образом менеджеры проектов могут использовать алгоритмический подход к оценке стоимости для анализа проектных характеристик. Опишите ситуацию, когда менеджеры выбирают подход, не основанный на принципе наименьшей стоимости проекта.

23.9. Реализуйте модель COCOMO с использованием программы электронных таблиц Microsoft Excel. Детальное описание этой модели можно загрузить с Web-узла COCOMO 2. На Web-странице данной книги я поместил ссылку на этот Web-узел.

23.10. Некоторые большие программные проекты требуют написания миллионов строк кода. Объясните, насколько полезными могут быть модели определения стоимости для таких систем. В каких случаях они могут быть неприменимы к большим системам?

23.11. Насколько этично назначить компанией-разработчиком относительно низкую цену для контракта, зная, что при столь неопределенных требованиях можно со временем повысить цену за дополнительные изменения в них, которые со временем обязательно будут сделаны заказчиком?

23.12. Следует ли менеджерам применять критерий производительности для определения деловой характеристики специалиста? Какие меры предосторожности необходимы, чтобы этот процесс определения профессиональных возможностей специалиста не влиял на качество его работы?

24 Управление качеством

Цели

Цель настоящей главы — представить основные принципы управления качеством и мероприятия, выполняемые в этой области. Прочитав эту главу, вы должны:

- знать основные процессы управления качеством, а также основные составляющие процесса управления качеством, а именно: обеспечение качества, планирование качества и контроль качества;
- понимать важность применения стандартов в процессе управления качеством;
- иметь представление о метрических показателях ПО и о различиях между прогнозируемыми и контролирующими показателями;
- понимать необходимость систем измерения при оценке показателей качества и знать об ограничениях в процессе измерения показателей ПО.

Содержание

Для многих организаций основным критерием деятельности является достижение высокого уровня качества производимой продукции либо предоставляемых услуг. В наше время невозможна поставка продуктов низкого качества, требующих устранения недоработок после доставки заказчику. К программным продуктам это относится не в меньшей мере, чем к таким промышленным товарам, как автомобили, телевизоры или вычислительная техника.

Однако качество программного продукта является достаточно сложным понятием, трудным для определения. Традиционно продукт считается качественным в том случае, если полностью соответствует техническим требованиям [82]. В идеале такое определение должно быть применимо ко всем продуктам, в том числе и к программным, однако здесь нас подстерегают некоторые проблемы.

1. Технические требования ориентированы на те свойства продукта, которые необходимы заказчику. Однако организация-разработчик может также иметь свои требования к разрабатываемому программному продукту (например, удобство сопровождения), которые обычно не включаются в технические требования заказчика.
2. Неизвестно, как точно определить и измерить определенные показатели качества (например, то же удобство сопровождения).
3. Как уже упоминалось в первой части книги, трудно создать полную спецификацию программного продукта. Поэтому, хотя созданный программный продукт будет полностью соответствовать спецификации, заказчик все равно может не получить высококачественного продукта.

Очевидно, необходимо прилагать усилия для совершенствования спецификации, однако на данном этапе следует смириться с тем, что она будет не лишена недостатков. Таким образом, следует признать существование проблемы несовершенства спецификаций и привести в действие ряд процедур для улучшения качества ПО в рамках ограничений, возникающих вследствие этой проблемы. Особенно это касается таких определяющих качественных характеристик программных продуктов, как удобство сопровождения, переносимость и эффективность, которые детально не определены в технических требованиях, однако оказываются критическими показателями для качества программных систем. В разделе 24.2, раскрывающем вопросы планирования качества, эти показатели рассматриваются более подробно.

Достижение необходимого уровня качества зависит от менеджеров по качеству компании-разработчика. Теоретически управление качеством основывается на принципе определения стандартов и процедурных норм, в соответствии с которыми должно разрабатываться программное обеспечение, а также на проверке выполнения этих норм всеми разработчиками. На практике, однако, понятие управления качеством имеет более емкое содержание.

Хорошие менеджеры по управлению качеством стремятся к созданию в компании атмосферы "культивирования качества", где каждый, кто занимается разработкой продукта, берет на себя обязательство достичь наивысшего уровня качества создаваемого продукта. Такие менеджеры стимулируют команду к качественному выполнению работы и к постоянному поиску идей повышения качества. При том что стандарты и процедурные нормы являются основой качества, опытные менеджеры по управлению качеством осознают значение тех неосязаемых аспектов качества программных продуктов, которые не могут быть включены в стандарты (например, изящество, читабельность и т.п.). Они поддерживают служащих, заинтересованных именно в таких нематериальных аспектах, а также поощряют профессиональное отношение к работе всех членов команды.

Процесс управления качеством состоит из трех основных видов деятельности.

1. *Обеспечение качества.* Определение множества организационных процедур и стандартов в целях создания ПО высокого качества.
2. *Планирование качества.* Выбор из этого множества соответствующего подмножества процедур и стандартов и адаптация их к данному проекту разработки ПО.
3. *Контроль качества.* Определение и проведение мероприятий, гарантирующих выполнение нормативных процедур и стандартов качества всеми членами команды разработчиков ПО.

Управление качеством предполагает возможность независимого контроля за процессом разработки ПО. Контрольные проектные элементы, получаемые в процессе разработки ПО, являются основой контроля качества. Они тщательно проверяются на соответствие стандартам и целям проекта (рис. 24.1.). Так как работы, выполняемые по обеспечению и контролю качества, в определенной степени независимы, это предполагает возможность объективного взгляда на процесс разработки ПО, благодаря чему руководство компании может своевременно получить информацию о проблемах или трудностях, которые возникают в работе над проектом.

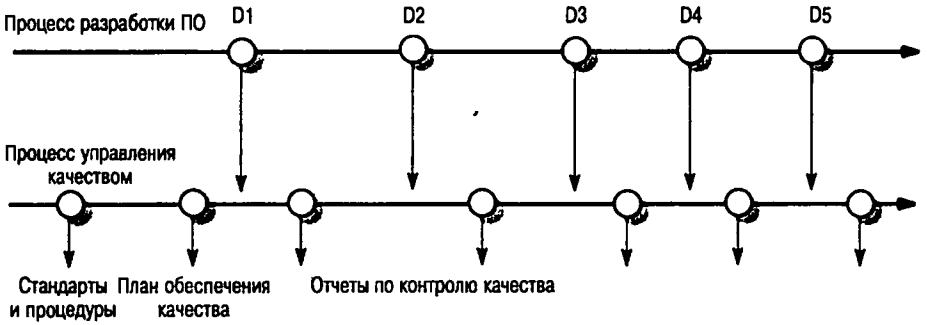


Рис. 24.1. Управление качеством и разработка ПО (буквой D обозначены контрольные проектные элементы)

Процесс управления качеством необходимо отделять от процесса управления проектом с тем, чтобы не ставить вопрос о компромиссе между качеством создаваемого ПО и бюджетом или графиком выполнения проекта. Над контролем качества должна работать независимая команда, которая отчитывается непосредственно руководству компании, минуя звено менеджера проекта. Команда контроля за качеством не должна быть также связана с группами разработки ПО, вместе с тем она берет на себя ответственность за качество на уровне всей организации.

Международно признанным стандартом, который любая компания в любых сферах производства может принять за основу развития системы управления качеством, можно назвать ISO 9000, разработанный Международной организации по стандартизации (ISO). ISO 9000 – это целый ряд всевозможных стандартов, применимых как в промышленности, так и в сфере услуг. ISO 9001 является наиболее обобщенным из этих стандартов и относится к организациям, занимающимся разработкой, производством и сопровождением различных товаров. Поддерживающая документация (ISO 9000-3) адаптирует ISO 9000 к разработке программных продуктов. Стандарт ISO 9000 описан во многих книгах, например [191, 265, 274, 9*, 20*].

Стандарт ISO 9001 является типовой моделью для процесса обеспечения качества. В этом нормативе описываются разнообразные аспекты данного процесса, а также определяются те стандарты и нормативы, которые должны быть приняты за основу производст-

венной деятельности компании. Так как процесс обеспечения качества не относится к разряду производственных видов деятельности, нормативы здесь детально не описаны. Любая организация, специализирующаяся на определенном виде услуг, должна самостоятельно провести детализацию своих нормативов и представить ее в специальном руководстве по управлению качеством.

В табл. 24.1 показаны те виды деятельности, которые охвачены в модели ISO 9001. Здесь у меня нет возможности пространно рассуждать о стандарте ISO и вникать в его глубины. Более подробное описание этой модели читатель найдет в книгах [178, 265], где можно получить сведения о применении данного стандарта к процессу управления качеством.

Таблица 24.1. Виды деятельности, охватываемые моделью обеспечения качества ISO 9001

За что отвечает меиеджмент	Элементы системы качества
Выявление изделий, не удовлетворяющих техническим требованиям	Контроль за разработкой изделий
Обработка, хранение, упаковка и доставка товара	Материально-техническое обслуживание
Товары, поставляемые заказчиком	Идентификация и отслеживание товара
Управление производственным процессом	Контроль и испытания готовой продукции
Оборудование для контроля и испытаний	Проведение обследования и тестирования
Проверка контракта	Корректирующая деятельность
Проверка документации	Отчеты об обеспечении качества
Внутренняя проверка качества	Обучение
Обслуживание	Статистические методы контроля за качеством

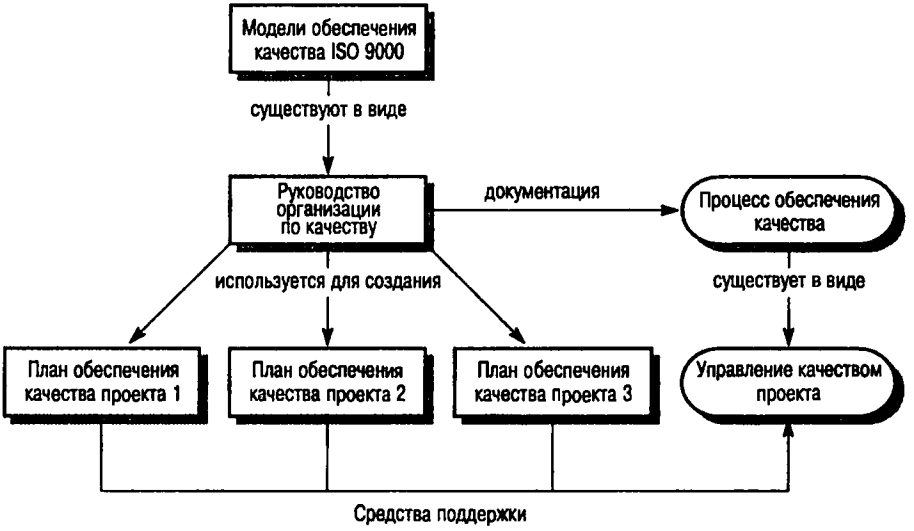


Рис. 24.2. Стандарт ISO 9000 и управление качеством

Нормативы по обеспечению качества занесены в специальное руководство, определяющее ход процесса по управлению качеством. В некоторых странах существуют специальные органы, подтверждающие соответствие процесса обеспечения качества, описанного в руководстве организации, стандарту ISO 9001. Более того, заказчики часто требуют от поставщиков сертификат по стандарту ISO 9001 как подтверждение того, насколько серьезно компания относится к изготовлению качественной продукции.

Взаимосвязь между ISO 9000, управлением качества и планами обеспечения качества отдельных проектов показана на рис. 24.2. Она заимствована из книги [178].

24.1. Обеспечение качества и стандарты

Деятельность по обеспечению качества направлена на достижение определенного уровня качества при разработке программного обеспечения. Она предполагает определение или выбор стандартов, применяемых либо к самому процессу разработки ПО, либо к готовому продукту. Эти стандарты могут быть частью процессов производства ПО. В ходе выполнения таких процессов могут применяться средства поддержки, учитывающие выбранные (или разработанные) стандарты качества.

В процессе обеспечения качества могут применяться два вида стандартов.

1. *Стандарты на продукцию.* Применимы к уже готовым программным продуктам. Они включают стандарты на сопроводительную документацию, например структуру документа, описывающего системные требования, а также такие стандарты, как, например, стандарт заголовка комментариев в определении класса объекта, стандарты написания программного кода, определяющие способ использования языка программирования.
2. *Стандарты на процесс создания ПО.* Определяют ход самого процесса создания программного продукта, например разработку спецификации, процессы проектирования и аттестации. Кроме того, они могут описывать документацию, создаваемую в ходе выполнения этих процессов.

Между стандартами на продукцию и стандартами на процесс существует очень сильная взаимосвязь. Стандарты на продукцию применимы к результату процесса разработки ПО, а стандарты на процесс в большинстве случаев подразумевают выполнение определенных действий, направленных на получение товара, соответствующего стандартам на продукцию. Более подробно этот вопрос рассматривается в разделе 24.1.2.

Стандарты в разработке программной продукции важны по целому ряду причин, основные из которых перечислены ниже.

1. Стандарты аккумулируют все лучшее из практической деятельности по созданию ПО. Как правило, практические знания приобретаются путем долгого поиска и ошибок. Привнесение этого опыта в определенный стандарт помогает избежать повторения прошлых ошибок. Стандарты в данном случае собирают знания и опыт, имеющие значение для организации-разработчика.
2. Стандарты предоставляют необходимую основу для реализации процесса обеспечения качества. Имея в наличии стандарты, обобщающие лучшие знания и опыт, для обеспечения качества достаточно контролировать, чтобы они выполнялись в процессе создания ПО.
3. Стандарты незаменимы, когда работа переходит от одного сотрудника к другому. В этом случае деятельность всех специалистов в организации подчиняется единому нормативу. Следовательно, требуется меньше затрат на изучение сотрудником новой работы.

Создание стандартов по разработке ПО – процесс долгий и утомительный. Такие национальные и международные организации, как Министерство обороны США, Американский национальный институт стандартизации (ANSI), Британский институт стандартов (BSI), НАТО, Институт инженеров по электротехнике и электронике (IEEE), специализируются на создании общих стандартов, которые могут применяться к широкому диапазону возможных программных проектов. Такие органы, как НАТО, или другие оборонные организации могут требовать соблюдения их собственных стандартов в контрактах на создание программного обеспечения.

Национальные (американские) и международные стандарты были разработаны для таких сфер программной деятельности, как терминология инженерии ПО, языки программирования типа Ada и C++, система обозначений, например для символов в схемах и чертежах, процедуры для разработки системных требований, деятельность по обеспечению качества, а также для аттестации ПО [177][1].

Группы обеспечения качества, которые занимаются составлением стандартов, обычно основывают нормативы организации на общих национальных и международных стандартах. Используя их в качестве отправного пункта, группа обеспечения качества разрабатывает свой "справочник" по стандартам. В нем содержатся стандарты, отражающие специфику деятельности данной организации. В табл. 24.2 приведены примеры стандартов, которые могут входить в состав такого справочника.

Таблица 24.2. Стандарты на продукцию и процесс разработки ПО

Стандарты на продукцию	Стандарты на процесс разработки ПО
Форма пересмотра архитектуры ПО	Руководство по проведению пересмотра архитектуры ПО
Структура документа, содержащего системные требования	Представление документации по нормативам ЕЭС
Формат заголовков программ и процедур	Процесс выпуска версии ПО
Стиль программирования языка Java	Процесс утверждения плана реализации проекта
Формат плана реализации проекта	Процесс контроля изменений
Форма запроса на изменения	Процесс регистрации выполнения тестов

Иногда специалисты по разработке ПО относятся к стандартам как к бюрократическому наследию, неприменимому к разработке ПО. Особенно это проявляется при выполнении такой утомительной и скучной (однако необходимой для соблюдения стандартов) процедуры, как заполнение всевозможных форм и регистрация работ. Конечно, с общей идеей полезности стандартов все согласны, однако при этом разработчики находят любую удобную причину, по которой именно для их проекта эти стандарты не так уж необходимы.

---

[1] *Отметим, что в Советском Союзе процесс создания ПО регламентировался стандартами ГОСТ ЕСПД (Единая система программной документации – серия ГОСТ 19.XXX). В настоящее время в России принят ряд стандартов создания автоматизированных систем, в состав которых входит программные компоненты: ГОСТ 34.601-90 "Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания" и др. [6*]. Однако эти стандарты имеют ряд недостатков, а некоторые их положения просто устарели. Поэтому отечественные разработчики ПО чаще используют современные международные стандарты. В Украине также принято несколько государственных стандартов на разработку ПО, в том числе ряд стандартов по обеспечению качества [3*].* – Прим. ред.

Чтобы не возникало подобных проблем в работе, менеджеры по качеству, отвечающие за разработку стандартов, должны быть достаточно подготовленными и действовать следующим образом.

1. Вовлечь самих программистов в разработку стандартов. Они должны ясно понимать, с какой целью разрабатывается стандарт, и четко следовать установленным правилам и нормативам. Важно, чтобы документ с описанием стандарта включал не только изложение самого норматива качества, но и объяснение необходимости именно этого норматива.
2. Регулярно просматривать и обновлять стандарты, чтобы идти в ногу с быстро развивающимися технологиями. Как только стандарт разработан, его помещают в справочник организации по стандартам, где его холят и лелеют, меняя с большой неохотой. Справочник по стандартам — вещь для организации необходимая, однако он должен развиваться по мере развития новых технологий.
3. Подумать о том, чтобы обеспечить поддержку стандартов программными средствами везде, где только можно. Всяческие "канцелярские" стандарты вызывают огромное количество жалоб, связанных с нудной и утомительной работой по их выполнению. Если же имеются в наличии средства поддержки, выполнение стандартов не требует больших усилий.

Стандарты на процессы разработки ПО также могут вызвать ряд проблем, если ставят перед командой разработчиков практнчески неосуществимые задачи. Такие стандарты дают руководящие советы по выполнению работы, при этом менеджеры проектов могут интерпретировать их каждый по-своему. Нет смысла указывать определенное направление работы, если оно неприменимо к данному проекту или к самой команде разработчиков. Поэтому менеджер проекта должен иметь право изменять стандарты процесса создания ПО в соответствии со специфическими условиями именно данного проекта. Здесь следует оговориться, что это утверждение не относится к стандартам на качество готовой продукции и на процесс сопровождения программной системы, которые могут быть изменены только после глубокого изучения данного вопроса.

Менеджер проекта и менеджер по управлению качеством могут легко избежать подводных камней, связанных с "неподходящими" стандартами, путем тщательной разработки плана мероприятий по обеспечению качества. Именно они должны решить, какие стандарты из справочника можно использовать без изменений, какие из них подлежат изменению, а какие следует исключить. Иногда возникает необходимость в разработке нового стандарта, что может быть вызвано условиями выполнения определенного проекта. Например, требуется установить стандарт для формальной спецификации, если прежде в проектах он не использовался. Такие стандарты должны разрабатываться в процессе выполнения проекта.

24.1.1. Стандарты на техническую документацию

Необходимость стандартов на документацию в программном проекте становится очевидной, если не существует никакого другого реального способа отображения процесса разработки ПО. Стандартные документы имеют четкую последовательную структуру, вид и качество, а значит, их легко читать и воспринимать. Существует три типа стандартов на документацию.

1. *Стандарты на процесс создания документации.* Определяют способ создания технической документации.

2. *Стандарты на документ.* Определяют структуру и внешний вид документов.
3. *Стандарты на обмен документами.* Гарантируют совместимость всех электронных версий документов.

Стандарт на процесс создания документации предоставляет описание способа изготовления документов. Он включает описания действий по созданию документов и предполагает программные средства для их создания. Кроме этого, нужно описать процедуры проверки и редактирования документов, благодаря которым обеспечивается необходимый уровень их качества.

Стандарты качества на процесс документирования должны быть достаточно гибкими, чтобы их можно было применять ко всем типам документации. Естественно, совсем необязательно проводить детальные проверки качества рабочей документации или служебных записок. Однако при работе с официальными документами, имеющими отношение к дальнейшей разработке продукта либо предназначенными для заказчика, нужно иметь установленную процедуру проверки их качества. На рис. 24.3 показана одна из возможных моделей такого процесса.

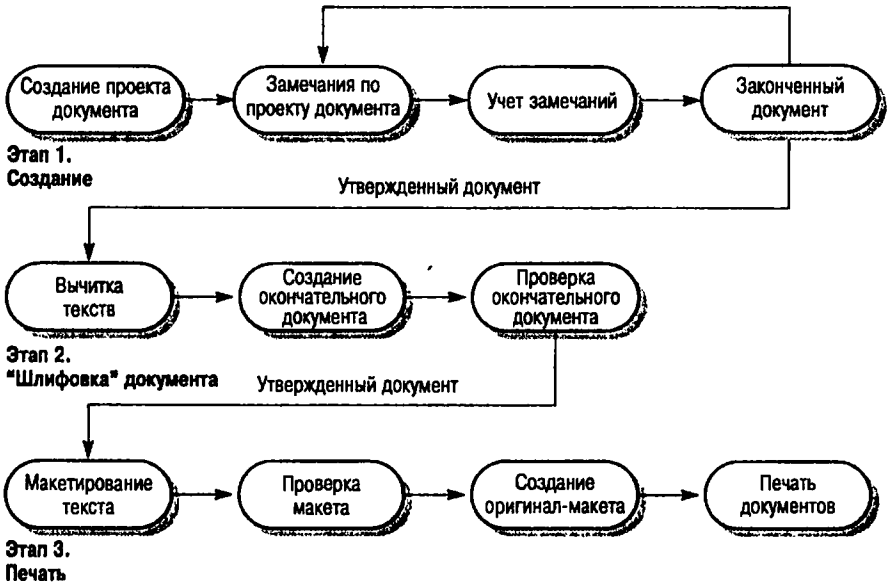


Рис. 24.3. Процесс создания документации, включающий проверку качества документов

Этапы сбора, учета и внесения замечаний в проект или очередную версию документа повторяются до тех пор, пока не будет создан документ соответствующего уровня качества. Уровень качества документа зависит от типа документации и от того, кому предназначен данный документ.

Стандарты на документацию следует применять ко всем документам, которые создаются в процессе работы над программным продуктом. Такие документы должны быть одинаковыми по стилю изложения и внешнему виду, а документы, которые относятся к одному типу, должны также иметь одинаковую структуру. Несмотря на то что стандарты на документацию могут быть адаптированы к определенного вида проектам, все же для организации неплохо было бы выработать собственный "фирменный" стиль, применяемый ко всем документам.

Приведем примеры различных стандартов на документацию.

1. *Стандарты идентификационных документов.* В процессе создания больших систем производятся тысячи документов, каждому из которых должно быть присвоено уникальное имя, т.е. каждый документ должен быть идентифицирован по определенной системе. Для формальных документов может применяться формальный идентификатор, определенный менеджером по конфигурации. Для неформальных документов идентификатор может быть определен менеджером проекта.
2. *Стандарты на структуру документа.* Каждый класс документов, создаваемый в процессе разработки ПО, должен иметь стандартную структуру. Стандартами на структуру определяются разделы, которые входят в документ, а также элементы форматирования и макетирования документа, например нумерация страниц, содержание верхнего и нижнего колонтитулов, нумерация разделов и подразделов.
3. *Стандарты внешнего вида документации.* Эти стандарты определяют "фирменный" стиль документов: использование шрифтов и стилей, логотипов, названия фирмы, цветовых выделений для элементов структуры документа и т.д.
4. *Стандарты на обновление документации.* Так как документы должны отображать изменения, возникающие в процессе разработки системы, желательно применять последовательный способ обозначения изменений в документации. Чтобы выделить новую, измененную версию документа, для печатных документов можно использовать обложки разных цветов, а для выделения изменений в тексте можно делать цветовые указатели на полях.

Особая роль отводится стандарту на совместимость документов, так как часто приходится работать с разными электронными версиями документов. Использование стандартов на совместимость позволяет осуществлять передачу документов в электронном виде и при необходимости восстанавливать их в первоначальном виде.

Поскольку стандартами на процесс создания документации предусмотрено использование инструментальных средств, стандарты на совместимость очерчивают сферу и способ использования таких средств в работе над документами. В качестве примера подобных стандартов на совместимость можно привести согласованный стандарт на комплект макросов при форматировании текста документа или на использование стандартной таблицы стилей (шаблонов) при работе с системами обработки текстов. Кроме того, введение в действие стандартов на совместимость может ограничить количество используемых шрифтов и стилей, что вызвано различными возможностями принтеров и компьютерных дисплеев.

24.1.2. Качество процесса создания ПО и качество программного продукта

Правило, лежащее в основе успешного управления качеством, гласит, что качество процесса производства ПО влияет и на качество готового программного продукта. Это предположение первоначально появилось в сфере промышленного производства, где качество продукции напрямую связано с качеством процесса ее изготовления. Разумеется, если говорить об автоматизированных системах массового производства, то должный уровень качества их работы автоматически обеспечивает высокое качество продукции. Такой подход показан на рис. 24.4.

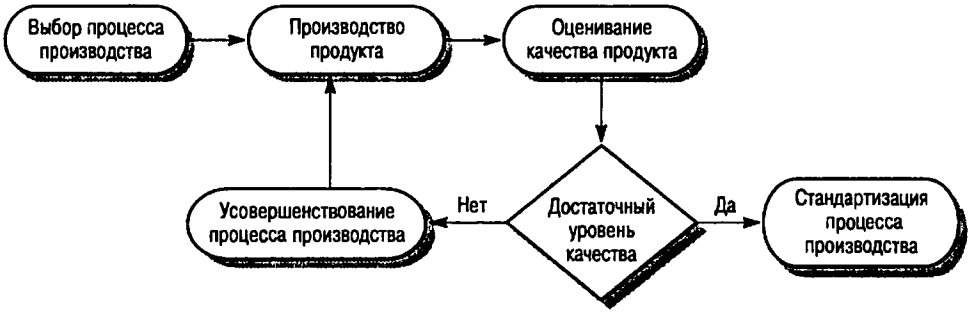


Рис. 24.4. Обеспечение качества продукции путем достижения должного уровня качества производственного процесса

В сфере разработки ПО качество процесса имеет особое значение. Это вызвано трудностью оценивания таких свойств программного обеспечения, как, например, удобство сопровождения, поскольку его можно оценить только после длительного периода использования программы. В данной области процесс улучшения качества начинается с изучения качественных продуктов и процессов их разработки, далее следует обобщение результатов исследования с последующим применением их в других проектах. Однако взаимосвязь между качеством процесса и качеством готового продукта не так проста, как кажется на первый взгляд. Изменение процесса производства не всегда ведет к появлению качественного программного продукта.

В промышленных отраслях связь между процессом производства и качеством продукта более очевидна, поскольку эти процессы относительно легко поддаются стандартизации и управлению. Сразу после проверки произведенных пробных изделий можно запускать серийное изготовление качественной продукции. Совсем другое дело с программным обеспечением, которое не изготавливается в прямом смысле этого слова, а разрабатывается. Поэтому процесс разработки ПО скорее относится к созидательным, нежели к механическим процессам, что многократно увеличивает значимость индивидуальных навыков и опыта разработчиков ПО. Кроме того, на качество программного продукта, независимо от использованного процесса создания ПО, могут оказывать влияние и внешние факторы, такие, как новизна проекта или коммерческое давление в целях более быстрого выпуска программного продукта.

Несмотря на все эти сложности, качество процесса создания ПО имеет большое значение для разработки качественных программных продуктов. Поэтому управление качеством включает в себя следующие функции.

1. Определение стандартов на процесс разработки ПО, например способ проведение проверок создаваемого ПО, времени, когда их следует проводить, и т.д.
2. Наблюдение над процессом разработки с тем, чтобы обеспечить выполнение стандартов.
3. Создание отчетности о ходе процесса разработки для менеджера проекта и заказчика программного обеспечения.

Самая большая сложность в обеспечении качества, основанном на стандартизации процесса создания ПО, состоит в том, что не всегда предписанный процесс создания подходит к данному типу разрабатываемого программного продукта. Например, стандарты на процесс создания ПО предусматривают завершение работы над требованиями и их утверждение до начала этапа написания программного кода. Однако для некоторых систем

может потребоваться прототипирование, что подразумевает написание соответствующих программ. Группа по обеспечению качества может предложить не выполнять прототипирования, так как все равно его качество невозможно проверить. В таких случаях требуется вмешательство старшего руководства компании, чтобы удостовериться в том, что процесс обеспечения качества служит поддержкой, а не помехой в выполнении проекта.

К вопросу о взаимосвязи между процессом создания ПО и качеством программного продукта вернемся в следующей главе, которая описывает методы совершенствования процесса создания ПО.

24.2. Планирование качества

Планирование качества нужно начинать на самой ранней стадии проекта. План обеспечения качества должен основываться на предполагаемых свойствах продукта, требуется также определить метод их проверки. Для этого необходимо определить понятие "должный уровень качества" программного продукта. Без этого программисты могут работать, делая акценты на разных свойствах продукта. Результат процесса планирования качества – это план обеспечения качества.

Из всех организационных стандартов в плане обеспечения качества должны быть отображены наиболее подходящие к создаваемому программному продукту или процессу его разработки. Если в проекте используются новые методы или инструментальные средства, то могут быть добавлены новые стандарты к уже существующим. В книге по управлению разработкой ПО [171], которую можно считать классической, предлагается приведенная ниже структура плана обеспечения качества.

1. *Представление продукта.* Описание продукта, намечаемый рынок его сбыта, а также ожидаемые свойства.
2. *Планы выпуска продукта.* Назначение крайних сроков выпуска версий программного продукта, распределение ответственности за разработку продукта и его обслуживание.
3. *Описания процессов.* Представление процессов разработки и обслуживания программного продукта в ходе выполнения проекта и управления им.
4. *Цели качества.* Планы и цели обеспечения качества продукта, включая описание наиболее важных его характеристик.
5. *Риски и управление рисками.* Описание основных видов риска, которые могут оказать влияние на уровень качества продукта, и мероприятия, направленные на снижение рисков.

При работе над планами обеспечения качества важно, чтобы они были как можно более краткими. Если документ будет слишком длинным, то специалисты вряд ли дочитают его до конца, что сведет на нет идею создания плана обеспечения качества.

Существует ряд потенциальных показателей качества продукта (табл. 24.3), которые должны учитываться при составлении плана обеспечения качества. На практике трудно создать настолько совершенную систему, чтобы она идеально удовлетворяла всем этим показателям, однако для конкретного проекта можно выбрать наиболее важные показатели и спланировать пути их достижения.

План обеспечения качества должен определять основные качественные показатели разрабатываемого продукта. Например, эффективность системы может иметь первостепенную важность по сравнению с другим показателями. Если это будет отображено в плане, то специалисты смогут найти компромисс между различными показателями системы. В плане также должен быть указан процесс оценивания уровня качества.

Таблица 24.3. Показатели качества программных продуктов

Надежность	Понятность	Переносимость
Безопасность	Возможность тестирования	Удобство эксплуатации
Безотказность	Адаптируемость	Возможность повторного использования
Устойчивость к внешним воздействиям	Модульность	Эффективность

План обеспечения качества должен определять основные качественные показатели разрабатываемого продукта. Например, эффективность системы может иметь первостепенную важность по сравнению с другим показателями. Если это будет отображено в плане, то специалисты смогут найти компромисс между различными показателями системы. В плане также должен быть указан процесс оценивания уровня качества.

24.3. Контроль качества

Контроль качества предусматривает наблюдение за процессом разработки ПО с тем, чтобы гарантировать соблюдение определенных нормативных процедур и стандартов. Как отмечалось ранее (см. рис. 24.1), контрольные проектные элементы проверяются согласно четко определенным стандартам процесса контроля качества.

Процесс контроля качества имеет собственный набор процедур и отчетов, которые могут быть использованы в процессе разработки ПО. Они должны иметь четкую структуру и быть понятными специалистам по разработке ПО.

Существует два взаимодополняющих подхода к процессу контроля качества.

1. Проверки качества, когда программный продукт, сопровождающая документация и процесс разработки анализируются группой проверяющих. Эта группа ответственна за проверку соблюдения стандартов проекта, а также за соответствие документации этим стандартам. Любые отклонения от стандартов регистрируются и подаются на рассмотрение менеджеру проекта.
2. Автоматизированная оценка ПО, когда программный продукт и его документация проверяются специальной компьютерной программой, которая сопоставляет их со стандартами данного проекта. В такую автоматизированную проверку можно включить количественный контроль некоторых характеристик ПО. Проблема измерения показателей качества ПО обсуждается в разделе 24.4.

24.3.1. Проверки качества

Проверки — это наиболее распространенный способ оценивания качества процесса разработки и создаваемого продукта. В проверку, как правило, включена группа специалистов, которые изучают отдельный этап или процесс разработки в целом, создаваемую систему и сопровождающую документацию для выявления возможных проблем. Результаты такой проверки обычно заносятся в официальный отчет и передаются разработчикам системы либо лицу, ответственному за исправление ошибок.

В табл. 24.4 представлены некоторые типы проверок. Инспектирование программ обсуждается в главе 19. Промежуточные проверки являются частью процесса управления проектом, который описан в главе 4. В этой главе проверки рассматриваются как часть

процесса управления качеством. В процессах различных проверок есть много общего, а описание организации такой проверки приведено в главе 19.

Таблица 24.4. Типы проверок

Тип проверки	Основная цель проверки
Инспекция структуры и программного кода системы	Выявить ошибки в требованиях, структуре и программном коде. Проверка проводится в соответствии с технологической картой возможных ошибок
Промежуточные проверки	Предоставить руководству отчеты о ходе выполнения проекта. Это может быть проверка и процесса разработки, и создаваемого продукта, а также выполнения бюджета и графика проекта
Проверки качества	Провести технический анализ компонентов продукта и документации с тем, чтобы найти несоответствия между спецификацией и структурой системы, программным кодом и документацией, а также гарантировать выполнение определенных стандартов качества

Целью команды проверки качества является обнаружение ошибок и противоречий с тем, чтобы довести их до сведения разработчика или автора документации. Все проверки основаны на документации, однако они не ограничены только спецификацией, структурой системы или программным кодом. Проверке подвергаются и такие документы, как модель процесса разработки, планы тестирования, процедура управления конфигурацией, стандарты процесса разработки и руководство пользователя.

В команду проверки качества должны быть включены те специалисты, которые могут принести наибольшую пользу. Например, при проверке структуры подсистем в такую команду должны входить программисты, которые участвовали в разработке подобных подсистем. Благодаря таким специалистам может появиться, например, новый взгляд на интерфейсы подсистем.

Ядром команды по проверке качества должно быть не более 3–4 человек, отобранных в качестве основных проверяющих. Один из них должен быть старшим, т.е. ответственным за принятие технический решений. Основные проверяющие могут приглашать других разработчиков проекта для участия в проверке. Им не обязательно принимать участие во всей проверке. Достаточно, чтобы они проверяли те части проекта, которые могут непосредственно повлиять на их работу. Кроме того, команда проверки качества может раздать тестируемый документ, чтобы получить письменные комментарии от других членов проекта.

Документы должны быть розданы до начала проверки с тем, чтобы рецензенты могли ознакомиться с ними и понять их суть. Хотя такие задержки и затягивают иногда процесс разработки, но проверка не будет иметь смысла, если команда не разберется в документации перед проведением проверки.

Сама проверка должна быть достаточно короткой (не более двух часов). Автор анализируемого документа должен "пройтись" по документу вместе с командой проверки. Один из членов команды должен руководить процессом проверки, а другой — записывать все ее результаты. Во время проведения проверки возглавляющий ее специалист ответственен за то, чтобы были рассмотрены все письменные замечания. По окончании проверки все действия проверяющих должны быть занесены в отчет, который подписывается проверяемым и лицом, возглавляющим проверку. В дальнейшем эти отчеты составят официальную документацию по проекту. Если были обнаружены лишь незначительные недос-

татки, проводить следующую проверку нет необходимости. Возглавляющий проверку специалист также несет ответственность за выполнение необходимых изменений в проекте. При необходимости внесения больших изменений нужно провести еще одну проверку.

24.4. Измерение показателей ПО

Измерения в области программного обеспечения — это получение числовых значений определенных показателей программного продукта или процесса его разработки. Значения этих показателей сравнивают между собой и со стандартами, применяемыми в данной организации. На основе этих сравнений можно сделать выводы о качестве продукта или процесса разработки. Например, организация планирует ввести новое программное средство для тестирования ПО. Перед тем как это средство будет введено в стандарт, нужно определить количество дефектов программной системы, обнаруженных за определенный период времени другими средствами тестирования. Затем следует повторить процесс тестирования новым средством. Если при этом будет обнаружено большее количество ошибок за то же время, значит, это средство окажется действительно полезным для проведения проверки правильности программного кода.

Ряд крупных компаний, таких, как Hewlett-Packard [135] и AT&T [22], ввели свои системы измерения показателей ПО и используют их в процессе управления качеством. Основное внимание в этих системах уделяется определению показателей, указывающих на присутствие дефектов в ПО, а также показателям, используемым в процессе проверки и аттестации программных продуктов. В работах [262, 150] рассмотрен вопрос введения таких систем в промышленное производство. В руководстве компании AMI [285] дается подробное описание такой системы измерений и ее применения для усовершенствования процесса разработки ПО.

Вместе с тем системы измерения показателей ПО пока не получили широкого распространения. Основной причиной является отсутствие очевидной пользы от этих систем — во многих организациях-разработчиках процесс создания программных продуктов все еще организован не лучшим образом и недостаточно развит для ведения подобных систем измерений. Кроме того, нет четких стандартов показателей ПО, отсюда ограниченный уровень технической поддержки по сбору и анализу подобных данных. Большинство компаний не готовы к введению систем измерений показателей ПО, поскольку не разработаны соответствующие стандарты и отсутствуют средства их поддержки.

Показатели программного обеспечения — это количественные показатели, которые можно измерить и которые характеризуют программную систему, процесс разработки ПО или сопровождающую документацию. Например, это может быть размер программного продукта, равный количеству строк кода, индекс непонятности [142], который является мерой читабельности письменного текста, количество зарегистрированных ошибок в программном продукте, а также количество человеко-дней, требующихся для создания системных компонентов.

Показатели делятся на два вида: контрольные и прогнозируемые. Контрольные показатели обычно соотносятся с процессом разработки ПО, а прогнозируемые — с готовым программным продуктом. Примером контрольного показателя (показателя процесса) может быть среднее значение затрат и времени, расходуемых на исправление зарегистрированных неполадок. В качестве примера прогнозируемого показателя можно привести

цикломатическую сложность программных модулей[2], среднюю длину идентификаторов в программе, а также количество атрибутов и операторов, относящихся к объектами системной структуры. На решения по управлению проектом могут оказать влияние как контрольные, так и прогнозируемые показатели (рис. 24.5). Подробнее показатели процесса создания ПО и их роль в совершенствовании этого процесса рассматриваются в главе 25. Здесь же внимание сосредоточено на показателях, прогнозирующих качество программного продукта.

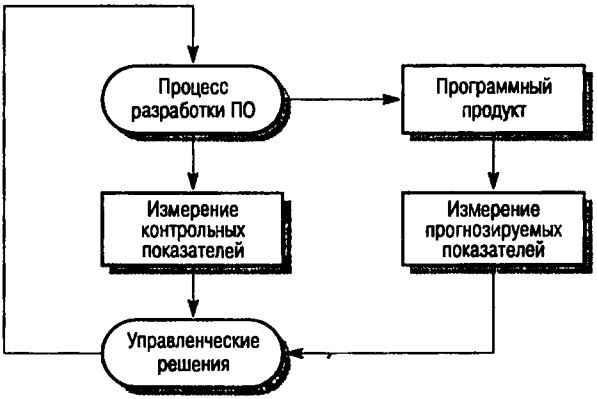


Рис. 24.5. Контрольные и прогнозируемые показатели

Часто невозможно провести прямое оценивание таких показателей качества программного продукта, как удобство сопровождения, сложность или понятность, поскольку они слагаются из самых разных факторов. Поэтому для их оценивания не существует прямой системы показателей. Отсюда следует вывод, что для начала необходимо измерить какой-либо внутренний показатель ПО (например, размер программы), а затем предположить, что существует взаимосвязь между тем, что мы можем измерить, и тем, что мы в действительности хотели бы узнать. В идеале между внутренними и внешними характеристиками программного продукта должна быть четкая прямая взаимосвязь, поддающаяся проверке.

На рис. 24.6 показаны внешние показатели качества, которые нас заинтересуют, и внутренние показатели ПО, которые можно измерить и соотнести с внешними свойствами. Между внешними и внутренними показателями ПО предполагается определенная взаимосвязь, однако неизвестно, какого вида эта взаимосвязь. Если необходимо определить внешние характеристики ПО путем измерения внутренних показателей, следует соблюсти три обязательных условия [198].

1. Точное и аккуратное проведение измерения внутренних показателей.
2. Наличие взаимосвязи между измеренными показателями и внешними поведенческими характеристиками ПО.
3. Обязательное изучение этой взаимосвязи, ее проверка и выражение в виде формулы или модели.

---

[2] *Цикломатическая сложность программного модуля характеризуется цикломатическим числом графа, отображающего структуру этого модуля (когда каждой точке ветвления программы соответствует вершина графа). Определение и формула вычисления цикломатического числа приведены в главе 20.* – Прим. ред.

Формулировка модели предусматривает определение вида функциональной зависимости между внешними и внутренними показателями (линейный, экспоненциальный или другой вид зависимости), что осуществляется путем анализа собранных данных, определения параметров, которые должны быть включены в модель, а также калибровки существующих данных. При построении такой модели необходим в первую очередь достаточный опыт работы со статистическими методами. Поэтому в такую деятельность следует вовлечь специалистов по математической статистике.

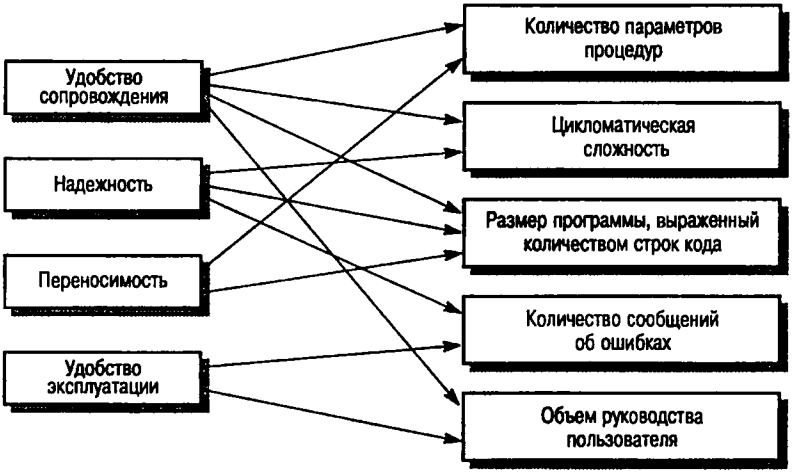


Рис. 24.6. Взаимосвязь между внутренними и внешними показателями ПО

24.4.1. Процесс измерения

Процесс измерения показателей ПО, который может быть частью контроля качества, показан на рис. 24.7. Каждый компонент системы анализируется отдельно, значения одинаковых показателей сравниваются между собой, а иногда и с аналогичными статистическими данными других проектов. Аномальные данные измерений по какому-либо системному компоненту должны стать поводом для проведения мероприятий по обеспечению качества этого компонента.

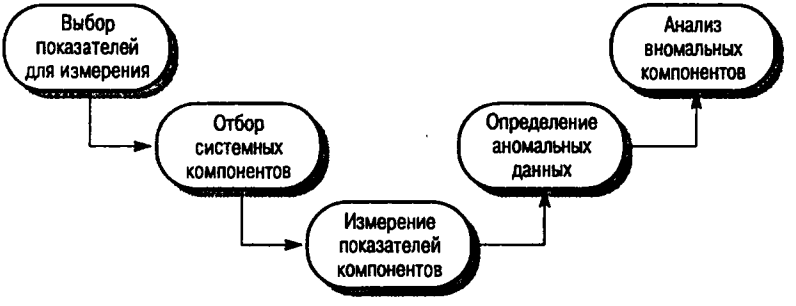


Рис. 24.7. Процесс измерения показателей программного продукта

Процесс измерений состоит из пяти основных этапов.

1. *Выбор показателей для измерения.* Для начала следует сформулировать те вопросы, на которые необходимо получить ответ посредством измерения, после чего определяются измеряемые показатели. Не нужно выбирать показатели, которые не соответствуют поставленным задачам. Парадигма "цель–вопрос–измерение" [26], которая рассматривается в следующей главе, является одним из лучших подходов к выбору измеряемых показателей.
2. *Отбор системных компонентов.* Часто совсем необязательно оценивать показатели всех компонентов программной системы. В одних случаях для анализа целесообразно сделать представительную выборку компонентов; в других достаточно оценить наиболее важные (критические) компоненты системы.
3. *Измерение показателей компонентов.* Это процесс измерения значений выбранных показателей для отобранных компонентов. Для этого обычно используются средства автоматического сбора данных. Они могут быть либо отдельными специальными средствами, либо встроенными в CASE-средства.
4. *Определение аномальных данных.* Значения измеренных показателей нужно сравнить между собой и с предыдущими измерениями, занесенными в базу данных. Важно отследить необычно высокие или, наоборот, низкие значения каждого вида показателей, так как компоненты с такими показателями могут быть причиной возникновения последующих проблем.
5. *Анализ аномальных компонентов.* Определив компоненты с аномальными показателями, их следует изучить для выявления возможного отрицательного влияния на качество программного продукта в целом. Например, аномальное значение такого показателя, как сложность компонента, не обязательно будет означать его плохое качество. Может быть другая причина для высокого значения этого показателя, которая не ведет к снижению качества.

Все собранные данные должны сохраняться в качестве ресурса организации, при этом обязателен анализ архивных данных по всем проектам, даже если они не используются в текущем проекте. После создания достаточно большой базы данных измерений на основе сравнения информации по различным проектам можно организовать специальную систему измерения показателей, которая отвечает нуждам данной организации.

24.4.2. Показатели программного продукта

Эти показатели отображают свойства программного продукта в целом. К сожалению, такие легко измеримые показатели, как размер и цикломатическая сложность программы, не имеют прямой и универсальной взаимосвязи с такими показателями качества, как надежность или удобство сопровождения. Эти взаимосвязи могут изменяться в процессе разработки продукта, зависят от технологии разработки и типа разрабатываемой программной системы. Организации-разработчики обязательно должны вести базу статистических данных. Именно такая база данных поможет в дальнейшем разобраться, какие свойства продукта соотносятся с теми показателями качества, в которых заинтересована организация.

Показатели программного продукта можно разделить на два класса.

1. Динамические показатели, которые измеряются в процессе выполнения программы.
2. Статические показатели, которые отражают статические представления системы, например структуру, программный код или документацию.

Эти два класса показателей соотносятся с разными свойствами продукта. Динамические показатели помогают в оценке эффективности и надежности программы, тогда как с помощью статических показателей можно оценить сложность, понятность и удобство эксплуатации программной системы.

Динамические показатели тесно связаны с показателями качества ПО. Относительно легко измерить время выполнения определенных функций и оценить время, необходимое для запуска системы. Именно эти показатели соотносятся с эффективностью системы. Таким же образом можно регистрировать количество системных ошибок и их типы, что напрямую связано с надежностью системы, как было описано в главе 16.

Статические показатели, как правило, имеют отдаленное отношение к качественным характеристикам ПО. Было предложено достаточное количество таких показателей, с которыми был проведен ряд экспериментов для выявления и оценки возможных взаимосвязей между этими показателями и такими свойствами, как сложность, понятность и удобство эксплуатации системы. В табл. 24.5 приведено несколько статических показателей, которые могут использоваться при оценке качества. Среди них объем программного кода и сложность управления являются наиболее надежными показателями для прогнозирования сложности, понятности и удобства сопровождения программного продукта.

Таблица 24.5. Показатели программного продукта

Показатели	Описание
Нагрузочный множитель по входу и нагрузочный множитель по выходу	Нагрузочный множитель по входу пропорционален количеству функций, которые вызывают другую функцию (назовем ее X). Нагрузочный множитель по выходу пропорционален количеству функций, которые вызываются функцией X. Высокие значения для множителя по входу означают, что функция X тесно связана с остальными компонентами системы и изменения в этой функции могут привести к существенным изменениям во всей системе. Высокое значение множителя по выходу означает высокую сложность самой функции X, которая вытекает из сложности логической схемы управления вызываемых ею компонентов
Объем программного кода	Этот показатель определяет размер программы. В общем случае, чем больше объем кода программного компонента, тем более сложным и подверженным ошибкам будет сам компонент
Цикломатическая сложность	Это мера сложности логической структуры программы. В свою очередь, сложность структуры программы связана с таким показателем, как понятность программного кода. Вычисление цикломатической сложности рассматривается в главе 20
Длина идентификаторов	Этот показатель измеряется как среднее длин идентификаторов в программе. Чем длиннее идентификаторы, тем понятнее, что они означают, а следовательно, более понятна и сама программа
Глубина вложения условных операторов	Здесь измеряется глубина вложений условных операторов в программе. Большая глубина вложения условных операторов затрудняет чтение программы, что ставит ее в разряд потенциально ошибочных программ
Индекс непонятности	Измеряется как среднее длин слов и предложений в документах. Чем выше показатель индекса непонятности, тем труднее понять документ

С начала 90-х годов был проведен ряд исследований для объектно-ориентированных показателей. Одни из них совпадают с показателями, приведенными в табл. 24.5, другие являются следствием объектно-ориентированного подхода к созданию ПО. Некоторые объектно-ориентированные показатели представлены в табл. 24.6. Они менее показательны по сравнению с аналогами из табл. 24.5, которые ориентированы на функциональный подход к созданию систем.

Таблица 24.6. Объектно-ориентированные показатели

Показатели	Описание
Глубина дерева наследования	Представляет количество дискретных уровней в дереве наследования, когда подклассы наследуют свойства и функции (методы) родительских классов. Чем глубже дерево наследования, тем сложнее структура системы
Метод ветвления по входу и выходу	Этот метод тесно связан с нагрузочными множителями по входу и выходу, описанными ранее, и означает, по существу, то же самое. Однако может оказаться полезным разделение вызовов от методов внутри объекта и вызовов от внешних методов
Взвешенные методы на класс	Означает количество методов, включенных в класс, средневзвешенных по сложности каждого метода. Таким образом, простой метод может иметь сложность 1, у более сложного метода будет более высокое значение сложности. Чем выше значение этого показателя, тем сложнее будет класс объектов. Сложные объекты более трудны для понимания
Количество игнорируемых операций (методов)	Это количество операций класса, которые игнорируются в подклассе. Высокое значение этого показателя говорит о том, что используемые классы плохо подходят в качестве родительских для данных подклассов

Множество показателей, выбираемых для измерения, зависят от проекта, целей команды управления качеством и типа разрабатываемого ПО. Все показатели, описанные в табл. 24.5 и 24.6, могут использоваться в тех или иных случаях. Однако бывают ситуации, когда эти показатели неприменимы. В этом случае компания-разработчик должна экспериментировать с различными показателями для поиска наиболее полно отвечающих ее потребностям.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Управление качеством ПО должно обеспечивать минимальное количество дефектов в разрабатываемой системе и соотаетствие стандартам удобства эксплуатации, надежности, переносимости и т.д. Деятельность по управлению качеством включает мероприятия по обеспечению качества, что создает необходимую базу стандартов по разработке ПО, планирование и контроль качества, где программный продукт проверяется на соответствие стандартам.
- Руководство по качеству любой организации должно формально представлять собой ряд нормативов по обеспечению качества. Оно может основываться на базовой модели стандартов ISO 9000.

- Стандарты в области разработки и сопровождения ПО аккумулируют лучшие практические приемы и знания. Процесс контролирования качества включает проверку того, соответствует ли разрабатываемое ПО определенным стандартам.
- Среди методик оценки качества наиболее широко используется проверка контрольных проектных элементов.
- Измерения показателей ПО проводятся для сбора количественной информации о программном продукте и процессе его разработки. Значения показателей ПО, которые получаются в результате измерений, используются для выводов о качестве продукта и процесса разработки.
- Показатели качества полезны для выявления аномальных системных компонентов, с качеством которых в дальнейшем могут возникнуть проблемы. При выявлении такие компоненты должны быть подробно исследованы.
- Для проверки качества программного обеспечения пока не существует стандартизированных и универсальных показателей. Выбор показателей и анализ измерений в значительной мере зависит от разрабатываемого программного продукта и наличия необходимых знаний и опыта у разработчиков ПО.

Упражнения

24.1. Объясните, почему высокий уровень качества процесса разработки должен привести к созданию высококачественного продукта. Опишите возможные проблемы в управлении качеством.

24.2. Из каких основных этапов состоит процесс проверки структуры ПО?

24.3. Дайте оценку качества ПО на основе показателей, приведенных в табл. 24.3. Рассмотрите каждый показатель в отдельности и объясните его значение.

24.4. Спроектируйте электронную форму, которую можно использовать для регистрации комментариев по проверке ПО и для рассылки их по электронной почте членам проверяющей команды.

24.5. Кратко опишите возможные стандарты, которые могут быть применены для следующих действий.

- Использование управляющих структур в языках программирования C, C++ или Java.
- Создание отчетов для срочных проектов.
- Процесс выполнения и утверждения изменений в программе (см. главу 29).
- Процесс покупки и установки нового компьютера.

24.6. Представьте, что вы работаете в организации, специализирующейся на разработке баз данных для микрокомпьютерных систем. Эта организация предполагает ввести систему измерений процесса разработки ПО. Предложите подходящие показатели для измерения, а также продумайте способ сбора необходимой информации.

24.7. Объясните, почему структурные показатели ПО не являются полноценными показателями для прогнозирования качества структуры системы.

24.8. Просмотрите соответствующую литературу и найдите другие показатели качества системной структуры, которые не описаны в этой книге. Изучите эти показатели и сделайте заключение о целесообразности их использования.

24.9. Мешают ли стандарты технологическим инновациям?

24.10. Программист высокого класса создает программный продукт с очень низким уровнем недоработок, однако упорно игнорирует организационные стандарты качества. Какой должна быть реакция менеджеров на подобное поведение?

25

Совершенствование производства ПО

Цели

Цель настоящей главы — познакомить читателя с тем, как усовершенствовать производство программного обеспечения, чтобы в итоге получать более качественный программный продукт. Прочитав эту главу, вы должны:

- знать основные принципы совершенствования производства ПО, а также понимать, насколько важен этот процесс;
- знать факторы, влияющие на качество программного продукта и производительность разработчиков ПО;
- понимать модель оценки уровня развития производства, которую можно применять для оценки качества технологии производства в больших организациях, разрабатывающих ПО;
- понимать, почему совершенствование на основе модели оценки уровня развития производства может быть применимо к производству не всех типов программного обеспечения.

Содержание

За последние несколько лет значительно возрос интерес специалистов, работающих в сфере инженерии ПО, к процессу совершенствования технологии разработки программных систем. Под совершенствованием технологии разработки программных систем подразумевается изучение существующего производства и внесение изменений в целях улучшения качества продукта и/или снижения издержек и времени разработки. Литература по данной тематике в основном рассматривает совершенствование производства как способ улучшения качества продукта, что подразумевает снижение количества недостатков в готовом продукте. Только после этого задачами изменения производства ПО становится снижение себестоимости или сокращение продолжительности работы над продуктом.

Как уже отмечалось в главе 24, между качеством продукта и качеством его производства существует тесная взаимосвязь. Считается, что повышение качества производства непосредственно влечет за собой получение более качественного продукта. Процесс разработки сам по себе достаточно сложен и состоит из целого ряда этапов. Как программный продукт, так и процесс его разработки, можно охарактеризовать (и оценить) определенными свойствами или параметрами; некоторые из них приведены в табл. 25.1.

Таблица 25.1. Параметры производства ПО

Параметр	Описание
Понятность	Четкое и доступное для понимания описание технологии разработки ПО
Наглядность результата	Оценка реализации проекта по ясным и ощутимым результатам
Наличие средств поддержки	Насколько производство ПО поддерживается CASE-средствами
Приемлемость	Степень одобрения и применяемости разработчиками данного процесса создания ПО
Безотказность	Организация работы таким образом, чтобы избежать или исправить те ошибки технологического процесса, которые вызовут неполадки в готовом продукте
Устойчивость к сбоям	Стабильное функционирование производства при возникновении непредусмотренных проблем
Удобство сопровождения	Способность технологического процесса к развитию при изменении системных требований и степень возможности внесения необходимых изменений
Скорость	Время, за которое будет завершен процесс разработки системы, соответствующий данной спецификации

Практически невозможно сразу довести до совершенства все без исключения параметры производства. Например, если необходимо быстро завершить разработку и ускорить работу, следует "снизить" параметр наглядности результата, поскольку это свойство требует регулярной подготовки документации через определенные интервалы времени, что замедляет процесс создания программного продукта.

Совершенствование производства — понятие сложное и означает не только выбор методов, средств или самой модели технологического процесса. Правы окажутся те, кто укажет на определенное сходство в деятельности всех организаций, разрабатывающих похожие типы программного обеспечения. Однако не следует забывать об организационных факторах,

нормативах и стандартах организации-разработчика, которые также влияют на ход производства. В процессе совершенствования производства всегда должна быть учтена специфика организации, более того, этот процесс должен стать частью ее деятельности.

Исходная модель процесса совершенствования производства показана на рис. 25.1. Этот процесс требует времени и осуществляется поэтапно. Каждый этап может иметь продолжительность до нескольких месяцев. Условием успеха совершенствования производства является приверженность организации поставленной цели, а также готовность вложить для ее реализации определенное количество ресурсов. Такая деятельность невозможна без утверждения высшего руководства организации и наличия бюджета, достаточного для проведения необходимых изменений в организации.

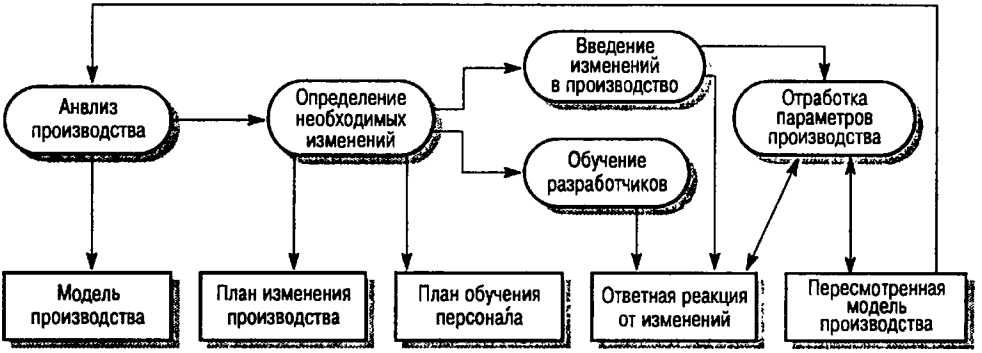


Рис. 25.1. Процесс совершенствования производства

Процесс совершенствования производства состоит из нескольких ключевых этапов.

1. *Анализ производства.* Этот этап включает исследование используемого процесса создания ПО и разработку его модели для подготовки и изучения соответствующей документации. В некоторых случаях можно ввести количественные параметры процесса, что позволит получить дополнительную информацию для исследования. Количественные измерения до и после внедрения изменений помогут дать более объективную оценку преимуществ или проблем, возникших в результате изменения производства.
2. *Определение необходимых изменений.* Здесь результаты анализа производства используются для определения критических параметров качества, графика работ и себестоимости, которые могут оказать влияние на качество готового продукта. Деятельность по совершенствованию производства должна сконцентрироваться на ослаблении влияния этих параметров путем выбора новых методов и средств технологического процесса.
3. *Введение изменений в производство.* На этом этапе новые методы и средства внедряются в существующий технологический процесс. Здесь особенно важно иметь достаточно времени для внесения изменений в технологический процесс, а также обеспечить совместимость этих изменений с существующими производственными процессами и организационными стандартами.
4. *Обучение измененному процессу производства.* Успешное изменение производства невозможно без предварительного обучения персонала, поскольку разработчики и менеджеры, ответственные за выполнение проекта, могут не принять изменений.

Часто изменения производства без необходимого обучения персонала приводят скорее к снижению качества продукта, чем к его повышению.

5. *Отработка параметров производства.* Введенные изменения никогда не дадут полного эффекта, на который вы могли бы надеяться. Вслед за реализацией нововведений должна произойти своего рода настройка, при которой обнаруживаются самые мелкие проблемы и неполадки и делаются необходимые поправки. Этот процесс может продолжаться в течение нескольких месяцев, пока разработчики не будут удовлетворены результатами.

После внесения одного или нескольких изменений процесс совершенствования можно повторить, включая этап анализа, определение "узких" мест и т.д. Это более практичный способ, чем введение всех изменений сразу. Кроме проблем в обучении, при многочисленных изменениях будет трудно оценить эффективность каждого из них.

25.1. Качество продукта и производства

Как было отмечено в главе 24, идея совершенствования производства основывается на том, что технологический процесс является критическим фактором, непосредственно влияющим на качество продукта. Автор этой идеи – американский инженер Деминг (W. E. Deming), который работал над проблемой улучшения качества в промышленности Японии после второй мировой войны. Японцы в течение долгих лет самоотверженно трудились над совершенствованием процесса производства. Это, несомненно, стало основой высокого качества японских товаров.

Деминг (и другие) ввел метод статистического контроля качества, который основан на простом подсчете недостатков продукта и соотнесении их с производственным процессом. Для устранения замеченных недостатков производственный процесс совершенствуется до тех пор, пока в конце концов не будут устранены все недоработки. Этим гарантируется предсказуемость результатов изменения процесса и снижение количества недоработок. Затем изменения проходят стадию стандартизации и начинается новый этап совершенствования.

В промышленности взаимосвязь "производство–продукт" прослеживается достаточно четко. Совершенствование производства само по себе ведет к повышению качества готовой продукции. Другое дело – неосязаемые и зависящие от интеллектуальной деятельности процессы производства ПО, которые к тому же не подлежат автоматизации. Ведь качество программного продукта зависит не только непосредственно от производственного процесса, но и от процесса проектирования, в котором главную роль играет человеческий фактор. Хотя для некоторых видов ПО ключевым в отношении качества является технологический процесс, однако при реализации новых задач люди, вовлеченные в процесс разработки, остаются ключевым фактором.

Существует четыре фактора, принципиальных для качества любой продукции, связанной так или иначе с интеллектуальной деятельностью, будь то программное обеспечение или же книги, фильмы и другие продукты, основой которых является проектирование. Эти факторы представлены на рис. 25.2.

Эти факторы могут оказывать различное воздействие, которое зависит от размера и вида программного проекта. Для очень больших систем, состоящих из отдельных подсистем и разрабатываемых разными командами программистов, технология разработки является ключевым фактором, влияющим на качество конечного продукта. Для таких проектов проблемными оказываются вопросы интеграции, управления и обеспечения взаимодействия. Так как разработка ПО продолжается несколько лет, состав рабочих групп

может меняться, кроме того, они состоят из специалистов с самыми разными возможностями и опытом. Состав группы может полностью поменяться за период выполнения проекта. Из этого следует, что влияние талантливых специалистов, имеющих особые навыки, не является доминирующим.

Рис. 25.2. Основные факторы, влияющие на качество программного обеспечения

Однако, если говорить о небольших проектах, где над реализацией работает несколько профессионалов, качество работы членов команды более важно, чем качественный технологический процесс. Если все члены команды имеют высокую квалификацию и достаточно опытны, более вероятно получение высококачественного продукта. В том случае, если команда не имеет навыков и опыта, качество технологического процесса может способствовать повышению качества продукции, однако не устранит в полной мере все погрешности и недоработки.

С другой стороны, в небольших командах хорошо разработанная технология имеет особое значение. Ведь члены команды не могут посвятить все свое время кропотливой бумажной работе. Наоборот, они проводят основную часть времени за разработкой и программированием системы, поэтому хорошие средства поддержки окажутся удачным подспорьем и повысят производительность труда. В больших проектах базовый уровень технологического процесса важен для управления потоком информации. Парадоксально, но при этом снижается влияние средств поддержки высокого уровня. Члены команды, стараясь понять различные компоненты системы, больше времени тратят на общение, чем на сам процесс программирования. Именно это, а не средства разработки, оказывается доминирующим фактором, влияющим на их производительность.

Факторы, положенные в основу прямоугольника на рис. 25.2, наиболее критичны. Качество продукта пострадает в любом случае, как только будет неправильно произведен расчет средств на выполнение проекта (независимо от его размера) либо будет представлен нереальный график работ, что сделает невозможным своевременное выполнение проекта. Для успешной реализации проекта необходимо достаточное количество материальных и временных ресурсов. В противном случае проект может спасти только высококвалифицированный и преданный персонал. И даже тогда при остром дефиците средств может пострадать качество конечного продукта.

Довольно часто оказывается, что причина появления на рынке программных продуктов низкого качества кроется отнюдь не в слабом управлении, плохо налаженном производстве или необученном персонале. Причина в другом: компании борются за выживание. Чтобы заполучить контракт на разработку программного продукта, многие компании дают заниженную оценку затрат. Назначение заниженной "выигрышной" цены (см. главу 23) — это необратимое следствие системы конкуренции. Поэтому неудивительно, что контроль качества товаров в таких условиях оказывается нелегкой задачей.

25.2. Анализ и моделирование производства

Анализ и моделирование производства ПО предполагает изучение используемых процессов разработки и их моделирование для определения ключевых параметров. Описание основных моделей процесса создания ПО представлено в главе 1. Частные варианты этих моделей постоянно используются в качестве примеров при рассмотрении отдельных тем, таких, как разработка требований, проектирование и т.д.. В книге [166] можно найти весьма удачный обзор темы моделирования процессов разработки ПО.

Анализ действующей технологии основан на ее тщательном изучении в целях выявления взаимосвязей между различными стадиями процесса производства. Первым шагом будет, несомненно, анализ качества, цель которого — определение основных параметров модели производства. Далее исследователь должен перейти к количественному анализу процесса разработки с использованием различных показателей, значения которых определяются автоматически или вручную. Только после этого технологический процесс можно представить в виде модели.

Анализ технологического процесса можно начать, принимая за основу его стандартную модель, которая существует в большинстве компаний или может определяться заказчиком. В таких стандартных моделях почти всегда определены основные этапы разработки и жизненный цикл создаваемых программных продуктов.

Стандартные модели удобны для начала анализа процесса создания ПО. Однако такие модели слишком обобщенные и предназначены лишь для общего описания основных этапов процесса разработки программного обеспечения. Всегда важно проникнуть в суть модели и выявить реально действующий процесс. Кроме того, действующий процесс может значительно отличаться от формальной модели.

В анализе производственного процесса используется два метода.

1. *Анкетирование и опросы.* Специалисты, работающие над программным проектом, заполняют анкеты, предоставляя информацию о реальном ходе проекта. Для официальной анкеты ответы уточняются во время индивидуальных интервью.
2. *Этнографические исследования.* В главе 6 уже обсуждалось, как этнографические исследования применяются для изучения процесса разработки ПО в контексте человеческой деятельности. Благодаря этому типу анализа раскрываются все тонкости и сложности, недостижимые для понимания в случае применения других методик.

Каждый из этих подходов имеет свои достоинства и недостатки. Если правильно составить вопросы, анкетирование поможет провести анализ сравнительно быстро. И наоборот, некорректно сформулированные или неверные вопросы приведут к созданию неправильной модели процесса. Кроме того, такой анализ имеет много общего с оцениванием. Поэтому опрашиваемые могут не сказать правду и даже дать те ответы, которые, по их мнению, будут желательны проводящему опрос.

Этнографический анализ на первый взгляд даст более верные результаты. Однако эта процедура достаточно длительная и кропотливая, она может продолжаться несколько месяцев. Кроме того, здесь данные основаны на внешнем наблюдении за процессом. Полный анализ происходящего должен проводиться с самой начальной стадии внедрения проекта и до поставки готового продукта заказчику и его последующего сопровождения. Это может оказаться непрактичным, так как график внедрения некоторых проектов растягивается на несколько лет. Поэтому этнографический анализ принесет больше пользы при детальном рассмотрении отдельных фрагментов процесса производства.

Результатом анализа можно считать модель производственного процесса любого уровня детализации. Модели процессов, приведенные в качестве примеров в этой книге, слишком абстрактны и упрощены, чтобы представить общую картину рассматриваемого процесса. На таком абстрактном уровне производственные процессы в разных организациях кажутся похожими. Однако, в зависимости от типа разрабатываемого программного обеспечения и рабочей среды организации, типовые модели детализируются и принимают определенный конкретный вид.

Для первоначального обсуждения процесса разработки типовая модель незаменима. Однако информации, которая в ней содержится, не достаточно для анализа процесса и внесения изменений. Для совершенствования процесса необходимы данные об этапах разработки, выходных результатах каждого такого этапа, персонале, качестве коммуникаций, графике работ и других компонентах деятельности организации по реализации проекта, так как все это может повлиять на создание ПО и должно быть учтено в модели производственного процесса. В табл. 25.2 описаны элементы, входящие в состав детализированной модели технологического процесса создания ПО (с указанием, как эти элементы изображаются на схеме модели).

Таблица 25.2. Элементы модели процесса создания ПО

Элемент модели	Описание
Действие (изображается прямоугольником с закругленными краями без тени)	Имеет четко поставленную цель, определены "вход" и "выход" действия. Примеры действий: подготовка тестовых данных для тестирования модулей, написание кода функции или модуля, редактирование документов и т.п. Обычно действие представляет собой элементарную часть этапа процесса, выполняется одним человеком или группой и не может быть разбито на более мелкие действия
Этап процесса (изображается прямоугольником с закругленными краями и с тенью)	Состоит из набора действий, связанных между собой и решающих общую задачу. Примеры этапов: анализ требований, проектирование системной архитектуры, планирование тестирования системы и т.д.
Контрольный проектный элемент (изображается как прямоугольник с тенью)	Это реальный результат действия или этапа, предусмотренный планом проекта
Условие (изображается параллелограммом)	Может быть либо предусловием и соблюдаться до начала выполнения этапа или действия, либо постусловием, которое должно выполняться после окончания действия или этапа
Роль (изображается в виде круга)	Очерчивает область ответственности лиц, работающих над проектом. Примеры ролей: менеджер по конфигурации, специалист по тестированию, проектировщик ПО и т.п. Несколько ролей могут выполняться одним и тем же специалистом, так же как и одна роль может быть разделена между несколькими лицами

Окончание табл. 25.2

Элемент модели	Описание
Исключение (в приведенном ниже примере не присутствует, изображается в виде ромба)	Это описание способов изменения технологического процесса в случае возникновения ожидаемых либо непредвиденных обстоятельств. В основном учитываются непредвиденные обстоятельства, поэтому присутствие этого элемента в модели зависит от изобретательности и предвидения менеджеров и разработчиков
Связь (изображается направленной стрелкой)	Отображает обмен информацией между людьми либо между человеком и компьютерными системами поддержки. Связи разделяются на формальные и неформальные. К формальным относится утверждение менеджером контрольного проектного элемента, примером неформальных связей может быть общение разработчиков по электронной почте для выяснения непонятных мест в документации

Модель также должна отображать нерабочие периоды времени в реализации проекта и взаимозависимость между этапами процесса разработки и контрольными проектными элементами, поскольку эти этапы можно выполнять как параллельно, так и последовательно. Все элементы проекта взаимосвязаны, например один и тот же специалист может быть задействован на разных этапах создания ПО. Контрольные проектные элементы могут зависеть от других контрольных элементов и даже от взаимоотношений между специалистами, работающими над проектом. Детализированные модели процесса создания ПО отличаются чрезвычайной сложностью. Создание единой модели, отображающей все аспекты производственного процесса, является крайне трудной и ответственной задачей.

Чтобы показать, насколько сложными бывают подобные модели, рассмотрим фрагмент процесса тестирования отдельного программного модуля большой системы, где используется процесс управления конфигурацией (см. главу 29). На рис. 25.3 показаны взаимоотношения между процессом тестирования, входом и выходом этого процесса, а также между пред- и постусловиями.

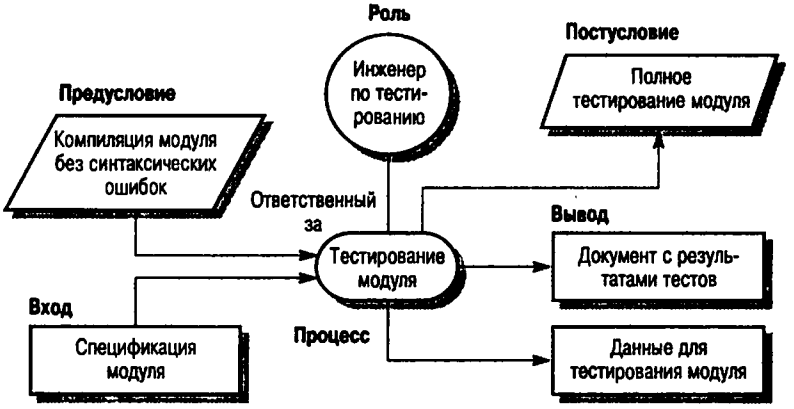


Рис. 25.3. Процесс тестирования программного модуля

На рис. 25.4 этап тестирования модуля разложен на составляющие действия. Этот фрагмент демонстрирует лишь малую часть общего процесса тестирования программных модулей. Четыре основных потока действий включают подготовку тестовых данных, написание тестовых процедур, проведение тестирования и создание отчетности о результатах тестирования.

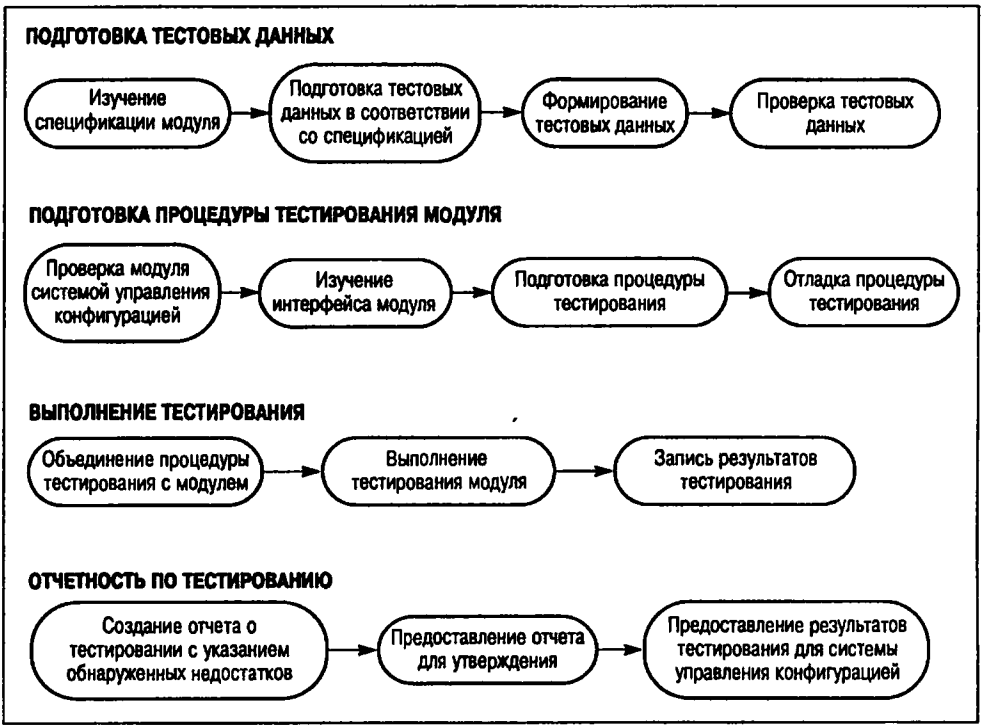


Рис. 25.4. Действия по тестированию модуля

Я намеренно не включил в эту модель пред- и постусловия и данные о входе и выходе этапа, так как это только усложнит модель и затруднит ее понимание. Не следует пытаться вместить всю информацию в одну-единственную модель. Лучше создать несколько моделей разного уровня обобщения и связать их какими-либо общими элементами, будь то действия или контрольные проектные элементы. Одни модели могут представлять этапы процесса производства, другие – данные, управляющие выполнением процесса.

25.2.1. Исключения в процессе создания ПО

Сам по себе процесс создания программного обеспечения – понятие достаточно сложное. Даже если организация разработала модель технологического процесса создания ПО, это еще не значит, что разработчики не столкнутся с какой-либо проблемой, не предусмотренной заранее. В действительности реагирование на такие непредусмотренные проблемы – часть повседневной работы менеджеров проекта. Поэтому идеальной представляется модель технологического процесса, которую можно легко изменять в ходе решения возникающих проблем. Ниже приведены примеры непредусмотренной ситуаций, к которым менеджер должен быть готов практически каждый день.

1. Ключевые разработчики могут заболеть и не выйти на работу как раз перед решающей проверкой проекта.
2. Невозможен обмен данными вследствие неисправности линий связи.
3. Вполне вероятно изменение структуры компании, в результате чего менеджеры будут тратить больше времени на решение организационных вопросов, вместо того чтобы работать над проектом.
4. Возможно вовлечение большего количества персонала в разработку нового проекта, имеющего больший приоритет, чем текущий.

В основном непредвиденные ситуации влияют на ресурсы, бюджет и график работ. Практически невозможно предусмотреть все экстремальные ситуации и включить их в организационную модель процесса. Поэтому в большинстве случаев менеджерам проектов приходится на ходу справляться с возникающими трудностями и вносить соответствующие изменения в стандартный процесс создания ПО.

25.3. Измерение производственного процесса

Измерение процесса создания ПО заключается в сборе количественных данных, характеризующих этот процесс. В книге [171], посвященной совершенствованию процесса создания ПО, утверждается, что сбор количественных показателей производственного процесса – необходимая составляющая его совершенствования. Эти показатели помогают узнать, насколько повысилась эффективность процесса после внесения изменений. Например, можно провести измерение времени и средств, потраченных на тестирование до изменения производства. Если совершенствование было эффективным, это должно снизить затраты, время или то и другое. Однако измерения сами по себе ничего не скажут об улучшении качества конечного продукта. Дополнительно необходимо иметь представление о значении показателей программного продукта (см. главу 24) и сопоставлении полученных результатов с новым процессом разработки.

Среди показателей процесса разработки выделим три вида.

1. *Время, потраченное на выполнение отдельного этапа работ.* Это может быть только рабочее время выполнения этапа, календарное время выполнения этапа или время работы отдельного специалиста.
2. *Ресурсы, необходимые для реализации этапа работ.* Ресурсы могут подсчитываться в человеко-днях, затратах на командировки либо ресурсах вычислительной техники.
3. *Количество повторений одного и того же события.* Среди таких событий можно назвать количество ошибок, обнаруженных при проверке программного кода, количество изменений в системных требованиях, среднее количество измененных строк кода и т.д.

Первые два вида показателей применяются для определения эффективности изменений процесса. Например, процесс разработки содержит несколько фиксированных временных точек, среди которых создание системной спецификации, завершение проектирования системной архитектуры, завершение формирования тестовых данных. При этом можно измерить время и средства, потраченные на переход от одной фиксированной точки к другой. Полученные результаты помогут определить те составляющие процесса, которые нуждаются в совершенствовании. После выполнения изменений следует измерить показатели процесса с тем, чтобы получить данные об эффективности изменений.

Измерение количества повторений одного события имеет более непосредственное отношение к качеству программного продукта. Например, обнаружение большего количества ошибок с помощью новой программы проверки, вероятно, улучшит качество готового продукта.

Основная проблема, связанная с измерением процесса создания ПО, — необходимость знать, что именно следует измерить. Для решения этого вопроса в статье [26] предлагается парадигма GQM (Goal-Question-Metric — цель-вопрос-показатель), с помощью которой определяется вид измерения и способ его использования. Эта парадигма основана "на трех китах".

1. *Цель.* Что является целью компании по совершенствованию процесса создания ПО? Это может быть повышение производительности труда программистов, сокращение времени разработки, повышение надежности готового продукта и т.д.
2. *Вопросы.* Это детализация поставленной цели. Как правило, каждая цель соотносится с рядом вопросов. Приведем примеры вопросов к указанным выше целям.
   - Как повысить количество строк кода, отлаженных программистом?
   - Как сократить время заключительного этапа разработки требований?
   - Как повысить эффективность проверки системы на надежность?
3. *Показатель.* Это та информация, которая поможет ответить на сформулированные вопросы, в частности определить, достигнуты или нет поставленные цели. Например, можно определить показатель производительности программистов, выражаемый в строках написанного ими кода, такой показатель, как уровень их профессионального опыта, показатель, равный количеству официальных контактов между заказчиком и исполнителем по каждому изменению системных требований, а также показатель количества проведенных тестов для выявления ошибок в системе.

Главное преимущество данного подхода состоит в том, что в нем разделены организационная деятельность (цели) и процесс производства (вопросы). Внимание концентрируется на сборе определенных данных и предусматривает разные способы анализа результатов измерения в зависимости от поставленного вопроса.

Подход GQM был объединен с описанной в следующем разделе моделью оценки уровня развития, разработанной Институтом инженерии программного обеспечения (США), что нашло свое воплощение в методе совершенствования процесса разработки ПО [285]. Разработчики этого метода предлагают поэтапный подход к совершенствованию процесса создания ПО. Этот подход основан на введении измерений только после того, как организация достигнет достаточного уровня развития технологии. Подход предлагает руководство и практические советы по внедрению измерений в целях совершенствования производства.

25.4. Модель оценки уровня развития

Институт инженерии программного обеспечения (Software Engineering Institute — SEI) при университете Карнеги-Меллон, получающий финансирование от Министерства обороны США, занимается распространением технологий создания ПО. Институт был учрежден с целью расширить возможности индустрии программного обеспечения в США, особенно для организаций, финансируемых Министерством обороны. В середине 80-х годов институт начал исследования по оценке возможностей компаний — разработчиков ПО. Особое внимание было уделено тем компаниям, которые подали заявки на финансирование проектов от Министерства обороны.

Результатом такой работы явилась модель оценки уровня развития в создании ПО. Модель оказалась чрезвычайно эффективным средством, сумевшим убедить сообщество программистов отнестись более серьезно к вопросу совершенствования процесса разработки ПО. Модель разбивает процесс создания ПО на пять этапов (рис. 25.5).

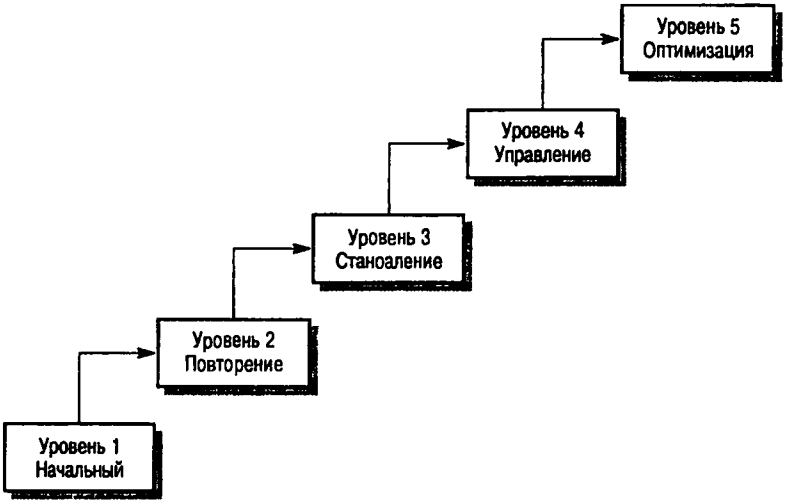


Рис. 25.5. Модель оценки уровня развития SEI

1. *Начальный уровень.* На этом уровне компания не имеет установленных процедур управления персоналом и планирования проектов. Даже если формальные процедуры для контроля над проектами и существуют, нет действующих механизмов проверки правильности использования этих процедур. Такая компания может успешно заниматься разработкой ПО, однако качество конечного продукта, а также ход выполнения проекта (соблюдение бюджета и графика работ) предсказать трудно.
2. *Уровень повторения.* В компании уже сложился определенный стиль управления, имеются системы управления качеством и конфигурацией. Название уровня обусловлено тем, что на данной стадии организация уже способна повторять проекты одного типа. Однако все-таки чувствуется недостаток организационной модели технологического процесса. Успех проекта зависит в основном от умения менеджеров вдохновить команду разработчиков и от возможности формализовать интуитивно понятный процесс разработки.
3. *Уровень становления.* Компания вводит стандартную технологию создания ПО, обеспечивая возможность совершенствования производственного процесса на количественном уровне. Приводятся в действие процессуальные нормы, которые гарантируют выполнение стандартного процесса во всех проектах.
4. *Уровень управления.* Компания уже имеет стандартную технологию создания ПО и программу сбора количественных данных. Накапливаются данные о показателях программных продуктов и процесса производства, которые впоследствии используются для совершенствования технологии разработки ПО.
5. *Уровень оптимизации*, которым предусматривается определенная приверженность организации-разработчика делу непрерывного совершенствования процесса создания ПО. Это означает, что совершенствование заложено в бюджете и плане деятельности организации и является ее неотъемлемой частью.

Первая версия модели оценки уровня развития подверглась жесткой критике за свою неопределенность. После приобретения опыта применения этой модели (см. следующий раздел) была разработана ее новая версия [273]. В ней были сохранены все пять уровней, однако теперь они более строго соотносились с ключевыми составляющими процесса создания ПО (рис. 25.6.). Результатом совершенствования процесса должно быть внедре-

ние в производство этих ключевых составляющих, а не достижение некоторого уровня, определенного моделью. Такой подход был использован при разработке модели оценки уровня развития системы управления требованиями [321].

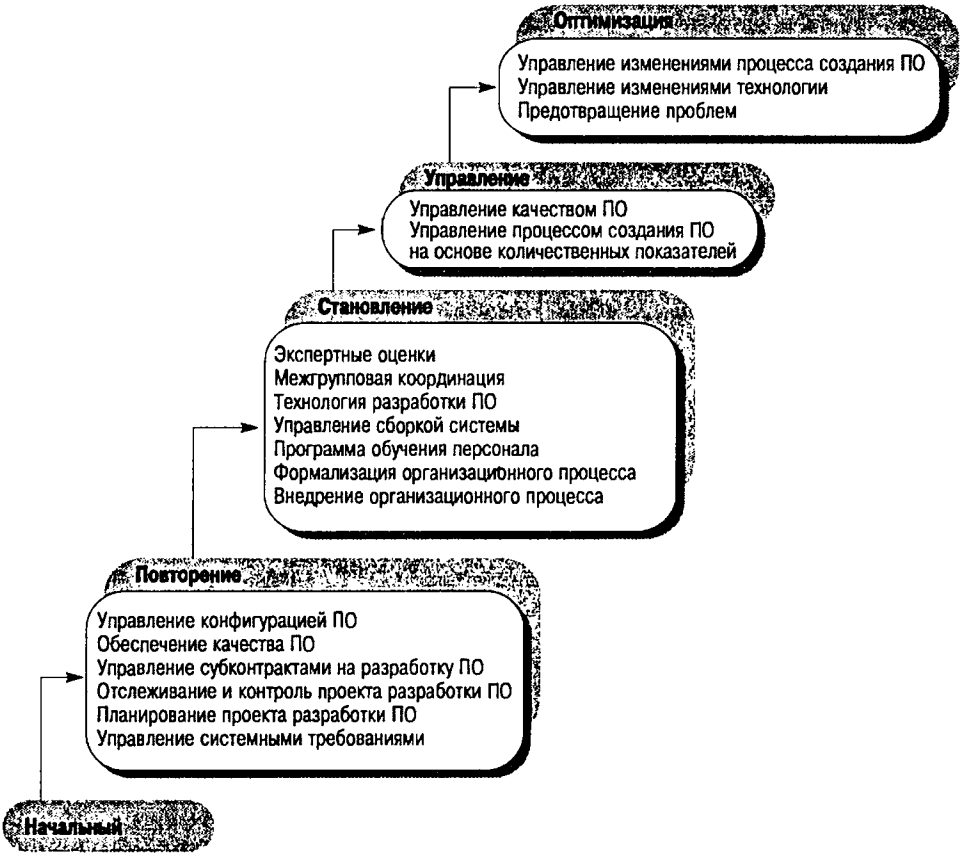


Рис. 25.6. Ключевые составляющие процесса создания ПО (©1993 IEEE)

Работа над моделью оценки уровня развития в SEI проводилась с учетом методов статистического контроля качества в промышленности. Допуская некоторое сходство между промышленным производством и производством программного обеспечения, я все же не думаю, что возможен простой перенос результатов, полученных в промышленности, в сферу разработки ПО. Как уже упоминалось в разделе 25.1, качество программного продукта может испытывать на себе влияние личностных факторов, например опыта и знаний программистов. В этой области деятельности такие факторы не менее важны, чем производственные.

Модель SEI оценки уровня развития достаточно весома и важна, однако я бы не советовал применять ее как единственную основу, определяющую весь процесс создания ПО. Она была предназначена в основном для компаний, которые занимаются разработкой программных систем для Министерства обороны, и служб, с ним связанных. Эти системы отличаются большими размерами и длительным сроком эксплуатации, а также сложностью интерфейса с аппаратным обеспечением и другими системами ПО. Над созданием таких систем работают достаточно большие команды программистов, которые должны подчиняться требованиям и стандартам, разработанным Министерством обороны США.

Первые три уровня модели SEI относительно легки для понимания. Среди основных составляющих процесса создания ПО есть и те, которые часто применяются в промышленном производстве. Некоторые организации-разработчики в состоянии достигнуть высших уровней модели, однако стандарты и процедуры, используемые на этих уровнях, не всегда понятны широкому кругу специалистов [95]. Известны случаи, когда успешно работающая компания не соответствует модели SEI, что вызвано обстоятельствами, присущими только данной организации.

Проблемы, с которыми организация может столкнуться на высших уровнях модели, никоим образом не ставят под сомнение полезность самой модели SEI. Однако на три основные проблемы все же стоит обратить внимание. Именно они могут снизить возможности организации в создании высококачественных продуктов.

1. Модель сосредоточена исключительно на управлении проектом, а не на процессе создания программного продукта. В модели не учтены такие важные факторы, как использование определенных технологий, например прототипирования, формальных и структурных методов, средств статического анализа и т.п.
2. В модели отсутствует анализ рисков и решений [51]. В главе 1 отмечалась важность оценки рисков, что позволяет обнаруживать проблемы прежде, чем они окажут воздействие на процесс разработки.
3. Не определена область применения модели, хотя авторы признают, что модель не является универсальной и подходящей всем организациям. Однако авторы не дают четкого разграничения организаций, которые могут или не могут внедрять модель в свою деятельность. Небольшие компании находят эту модель слишком бюрократичной. В ответ на эту критику были разработаны стратегии совершенствования технологического процесса для малых организаций [174].

В работе [271] проведено сравнение модели оценки уровня развития со стандартами ISO 9000 управления качеством, рассмотренными в главе 24. Здесь каждая составляющая процесса из модели оценки уровня развития сравнивается с требованиями процесса управления качеством (см. табл. 24.1.). В большинстве случаев была отмечена взаимосвязь между составляющими процесса модели SEI и требованиями ISO 9000. Модель SEI более детализирована и предоставляет основу для совершенствования процесса создания ПО, что не предусмотрено в стандартах ISO 9000. Организации, которые дошли до второго или третьего уровней развития по модели SEI, более других соответствуют стандартам ISO 9000. Однако в связи с определенной абстрактностью стандартов ISO 9000 организации и на первом уровне развития могут претендовать на соответствие этим стандартам.

Институт SEI разработал несколько других моделей оценки уровня развития, например: модель оценки уровня развития процесса разработки программных систем (см. главу 2), модель оценки приобретения, предназначенная для оценки приобретенного организацией программного и аппаратного обеспечения, модель оценки уровня развития персонала (см. главу 22) и др.

25.4.1. Оценивание уровня развития

Говоря о модели SEI оценки уровня развития, иногда забывают о том, что главной целью создания модели было намерение Министерства обороны США оценить возможности поставщиков программного обеспечения. На данный момент пока не существует четких требований к достижению определенного уровня развития организаций-разработчиков. Однако принято считать, что у организации, достигшей высокого уровня, больше шансов выиграть тендер на поставку ПО. В будущем от компаний потребуется определенный уровень развития (скорее всего, это будет третий уровень по модели SEI) для того, чтобы участвовать в тендерах Министерства обороны.

Модель оценки уровня развития ориентирована в основном на оценивание всей организации, а не отдельных проектов. С точки зрения разработчиков, это справедливо. Если организация находится, скажем, на первом уровне модели, это совсем не значит, что ее проекты должны котироваться так же низко. Отдельные команды программистов могут работать на более высоком уровне.

Основой оценивания уровня организации-разработчика является анкета, которая должна определить ключевые процессы в деятельности организации. Она заполняется путем опроса менеджеров разных проектов. Ответы, занесенные в анкету, анализируются, после чего выводится общий счет. Модель процесса оценивания показана на рис. 25.7.

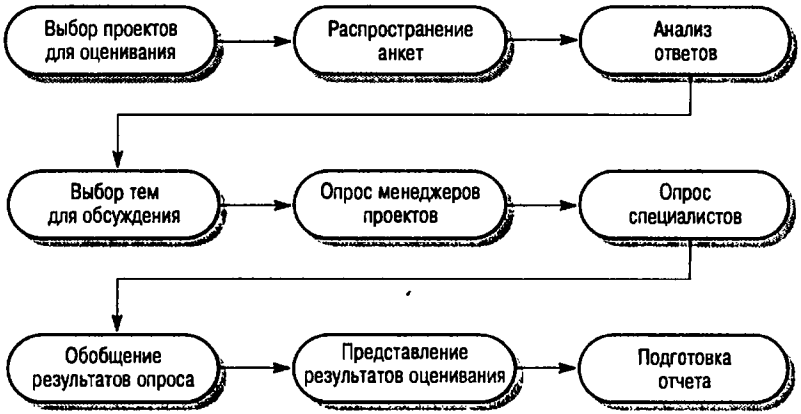


Рис. 25.7. Процесс оценивания уровня развития организации

Основной недостаток этого процесса оценивания, на наш взгляд, состоит в чрезмерном внимании к достижению высокого уровня. Организация обязана выполнить все предписанные моделью действия на определенном уровне, только тогда ей будет присвоен этот уровень. Даже если организация достигла 80% по второму уровню и 70% по третьему, все равно она будет котироваться первым уровнем. Здесь уместен более структурированный подход, учитывающий применяемые в данной организации виды деятельности и стандарты.

Метод оценивания уровня развития и совершенствования производственного процесса SPICE [272, 106] отличается большей гибкостью. Этот метод был предложен в качестве основы для улучшения стандарта ISO. В нем были сохранены уровни модели SEI, но добавлены другие ключевые виды деятельности (например, процесс "заказчик–поставщик"), которые проходят через все уровни.

Целью проекта Bootstrap было расширение и адаптация модели SEI к более широкому кругу организаций. В результате появилась одноименная модель [146, 206], в которой сохранены основные принципы модели SEI, однако имеется ряд нововведений.

- Предлагается система управления качеством для поддержки процесса совершенствования производства ПО.
- Проведено разграничение организации, методологии и технологии.
- Предложена базовая модель процесса разработки ПО (создана по образцу модели, используемой в Европейском аэрокосмическом агентстве).

Рассматриваемые модели оценки развиваются с учетом всех альтернативных подходов. Я даже предполагаю, что, когда выйдет эта книга, уже будет создана новая версия модели оценки уровня развития, в которую войдут лучшие идеи, имеющиеся в других моделях. Будем надеяться, что именно она станет признанным стандартом для оценивания и совершенствования процесса создания ПО.

25.5. Классификация процессов совершенствования

Как уже отмечалось, классификация процессов совершенствования производства программного обеспечения по модели SEI больше подходит для процесса разработки больших и длительно эксплуатируемых систем, создаваемых большими компаниями-разработчиками. Для малых и средних компаний-разработчиков данная модель подходит не в полной мере.

Вместо того чтобы разбивать процесс совершенствования производства на уровни и строить между ними нестойкие взаимосвязи, рациональнее, по моему мнению, применить обобщенную классификацию процессов совершенствования производства, которая подходит большинству организаций и программных проектов. Можно выделить несколько общих типов процессов совершенствования.

1. *Неформальный процесс.* Не имеет четко выраженной модели совершенствования производства. Его с успехом может использовать отдельная команда разработчиков. Неформальность процесса никоим образом не исключает такие формальные действия, как управление конфигурацией; однако при этом сами действия и их взаимосвязи не предопределены заранее.
2. *Управляемый процесс.* Имеет подготовленную модель, которая управляет процессом совершенствования. Модель определяет действия, их график и взаимосвязи между ними.
3. *Методически обоснованный процесс.* Подразумевается, что введены в действие определенные методы (например, систематически применяются методы объектно-ориентированного проектирования). Для процессов этого типа будут полезными CASE-средства поддержки проектирования и анализа процессов.
4. *Процесс непосредственного совершенствования.* Имеет четко поставленную цель совершенствования технологического процесса, для чего существует отдельная строка в бюджете организации и определены нормы и процедуры внедрения нововведений. Частью такого процесса является количественный анализ процесса совершенствования.

Эту классификацию не назовешь четкой и исчерпывающей — некоторые процессы могут одновременно относиться к нескольким типам. Например, неформальность процесса является выбором команды разработчиков. Эта же команда может выбрать определенную методику разработки, имея при этом все возможности непосредственного совершенствования процесса. Такой процесс подпадает под классификацию *неформальный, методически обоснованный, непосредственного совершенствования.*

Необходимость приведенной классификации обусловлена тем, что она предоставляет своего рода костяк для комплексного совершенствования технологии создания ПО и дает возможность организации выбирать разные типы процессов совершенствования. На рис. 25.8 показаны соотношения между разными типами программных систем и процессами совершенствования их разработки.

Знание типа разрабатываемого продукта сделает соответствие между программными системами и процессами совершенствования, показанное на рис. 25.8, полезным при выборе процесса совершенствования. Например, требуется создать программу поддержки перехода ПО с одной компьютерной платформы на другую. Такая программа имеет достаточно короткий срок эксплуатации, поэтому в ее разработке не требуются стандарты и специальное управление процессом совершенствования, как при создании долгоживущих систем.

Многие технологические процессы в наше время имеют CASE-средства поддержки, поэтому их можно назвать *поддерживаемыми процессами.* Методически обоснованные процессы поддерживаются инструментальными средствами анализа и проектирования. Эффективность средства поддержки зависит от применяемого процесса совершенствования. Например, в неформальном процессе могут использоваться типовые средства поддержки

(средства прототипирования, компиляторы, средства отладки, текстовые процессоры и т.п.). Вряд ли в неформальных процессах будут использоваться на постоянной основе более специализированные средства поддержки.

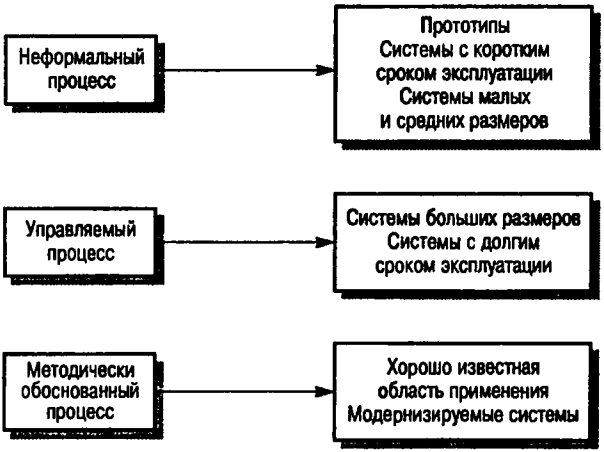


Рис. 25.8. Применимость процессов совершенствования

На рис. 25.9 показан широкий спектр средств поддержки, применяемых при разработке ПО. Эффективность отдельных средств напрямую зависит от типа выбранного процесса совершенствования. Например, с финансовой точки зрения только инструментальные средства анализа и проектирования подойдут для сопровождения методически обоснованного процесса. Специализированные средства применяются для совершенствования определенных процессов разработки ПО.

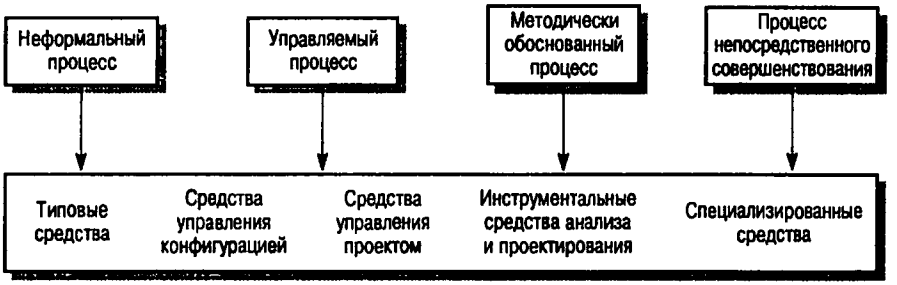


Рис. 25.9. Средства поддержки процессов совершенствования

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Процесс совершенствования разработки ПО включает следующие этапы: анализ производства ПО, стандартизацию, измерение количественных показателей и внесение изменений в технологию разработки ПО. Необходимой составляющей процесса совершенствования является обучение персонала.
- Модели процесса совершенствования состоят из описания действий, этапов, ролей, исключений, связей, контрольных элементов.

- Измерения количественных показателей проводятся для выяснения определенных вопросов относительно разрабатываемого ПО и основываются на организационных задачах по совершенствованию процесса разработки.
- Согласно модели оценки уровня развития SEI процессы совершенствования классифицируются как начальные, повторения, становления, управления и оптимизации.
- Модель оценки уровня развития SEI была разработана и подходит в основном для больших программных систем, разрабатываемых большими командами специалистов. Эта модель обязательно должна адаптироваться к условиям конкретной организации-разработчика.
- Классификация процессов совершенствования производства ПО подразделяет их на неформальные, управляемые, методически обоснованные и непосредственного совершенствования. Такая классификация полезна для определения средств поддержки выбранного процесса совершенствования.

Упражнения

25.1. Создайте модели следующих процессов.

- Разжигание костра.
- Приготовление комплексного обеда (меню по вашему выбору).
- Написание небольшой программы (не более 50 строк).

25.2. В каких условиях качество работы команды разработчиков будет определять качество готового программного продукта? Приведите примеры программных систем, при разработке которых особое значение имеют талант и способности отдельного программиста.

25.3. Предположим, что целью совершенствования процесса разработки ПО является увеличение количества повторно используемых компонентов. Сформулируйте три вопроса в соответствии с парадигмой "цель-вопрос-показатель".

25.4. Опишите три типа числовых показателей, которые могут использоваться в качестве данных для совершенствования процесса разработки ПО. Приведите по одному примеру каждого типа показателя.

25.5. Назовите два главных достоинства и два основных недостатка концепции оценивания и совершенствования процесса разработки ПО, которая положена в основу модели оценки уровня развития SEI.

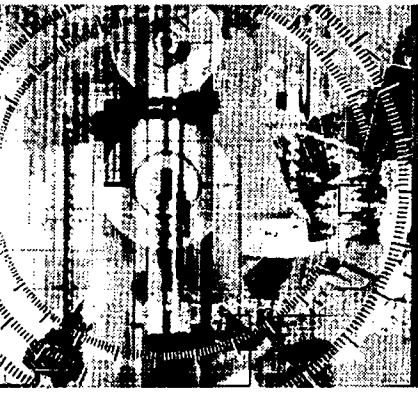
25.6. Приведите две области применения ПО, для которых модель SEI неприменима. Обоснуйте ваш выбор.

25.7. Рассмотрите технологический процесс создания ПО в вашей компании и определите его тип. Сколько ключевых составляющих процесса по модели SEI находятся в применении? Какого уровня развития достигла ваша компания по этой модели?

25.8. Приведите пример трех средств поддержки, которые могут использоваться для сопровождения процесса совершенствования производства ПО.

25.9. Объясните, почему методически обоснованный процесс не обязательно должен быть управляемым.

25.10. Считаете ли вы унижающими достоинство программы совершенствования производства, которые предусматривают оценивание работы персонала компании? Что, по-вашему, вызовет сопротивление введению такой программы в действие?



Часть VII

Эволюция программного обеспечения

26 Наследуемые системы

Цели

Цель настоящей главы — ввести читателя в тему наследуемых систем и показать структуру таких систем. Прочитав эту главу, вы должны:

- иметь понятие о наследуемых системах и о причинах особого значения этих систем для развития компаний во многих сферах бизнеса;
- знать общие структуры наследуемых систем;
- понимать принципы функционально-ориентированного проектирования, на котором основывается большинство стратегий по разработке наследуемых систем;
- уметь оценивать наследуемые системы с различных точек зрения.

Содержание

Для приобретения программного обеспечения компании обычно должны выложить немалую сумму. Естественно, чтобы оправдать затраты, программные продукты должны находиться в использовании по крайней мере несколько лет. Жизненный цикл программ может быть самым разным, но, как правило, большие системы успешно функционируют и более десяти лет. Некоторые компании не отказываются и от таких систем, которым уже по 20 лет и более. От многих из них все еще зависит деятельность крупных компаний, и малейшая ошибка в системе приводит к сбою их деловой активности. Именно такие системы и получили название *наследуемых* (legasy system).

Это, конечно же, не те старые системы, которые были разработаны и запущены в самом начале эры вычислительной техники. Деловая активность находится в постоянном развитии, обусловленном многими факторами, в том числе эволюцией экономики на национальном и международном уровнях, изменением рынка, законов, менеджмента и структурными преобразованиями. Эти факторы служат причиной появления новых или изменения существующих требований к программному обеспечению, что непосредственно приводит к модификации существующих систем. За несколько лет наследуемая система может пережить целый ряд подобных изменений, совершаемых разными специалистами, поскольку одному человеку практически невозможно полностью освоить систему во всех аспектах.

Предприниматели, как правило, настроены на систематическое обновление и модернизацию существующего оборудования. Однако, что касается списания и замены наследуемых систем современным ПО, такие действия могут повлечь за собой серьезный риск и необратимые последствия в деятельности компаний. Как отмечалось в главе 4, многие менеджеры не желают рисковать, устанавливая неизвестное и непредсказуемое современное ПО. Замена наследуемой системы – дело рискованное по многим причинам.

1. Редко можно найти такую наследуемую систему, которая имеет полное и точное техническое описание. Старое описание может быть утеряно, а если оно и существует, вряд ли там будут указаны все изменения, сделанные в системе. Поэтому трудно сравнивать технические характеристики и функциональные возможности старой системы с характеристиками ее возможной преемницы.
2. Функционирование наследуемой системы тесно связано с деловой активностью компании. При замене системы деятельность компании также претерпит изменения, что может привести к непредсказуемым расходам и необратимым последствиям.
3. Некоторые встроенные в систему правила, регулирующие область торгово-промышленных отношений компании, могут быть нигде не документированы. Эти правила обеспечивают своеобразные рамки, в которых должна вестись коммерческая деятельность, и нарушение этих рамок окажет не самое лучшее влияние на развитие бизнеса. Например, страховая компания могла включить в систему правила оценки риска страховых полисов. Если эти правила не выполняются, компания может быть вовлечена в работу с полисами высокой степени риска, что в дальнейшем вызовет большие затраты на выплату страховых возмещений.
4. Создание новых программных систем связано с риском, так как новизна системы подразумевает появление непредусмотренных проблем. Поставщик ПО может доставить программный продукт не вовремя, либо может измениться его цена.

Использование наследуемых систем избавляет организацию от риска, связанного с их заменой. Однако модернизация старой системы становится дороже с каждым годом эксплуатации. Возрастающая стоимость модернизации системы, находящейся в эксплуатации несколько лет, определяется многими факторами.

1. Отдельные части системы разрабатывались разными командами программистов, поэтому в них отсутствует единство стиля программирования.
2. Система либо ее отдельные части могут быть написаны с помощью языков, давно вышедших из употребления. Трудность подбора специалистов, знающих эти языки, усугубляется дорогостоящими договорами на внешнее сопровождение системы.
3. Документация системы часто бывает устаревшей и не отвечает современным требованиям. Иногда единственной документацией может остаться исходный код программ. В особо трудных случаях исходный код давно утерян, и все, что доступно, — это функционирующая версия программы.
4. Долгие годы эксплуатации могут оказать разрушительное воздействие на систему и исказить ее настолько, что она станет практически недоступной для понимания. Кроме того, в систему могут быть добавлены либо подогнаны к ней другие программы.
5. Система может быть оптимизирована для экономного использования памяти или для быстрого выполнения. Это создает дополнительные трудности для программистов, владеющих современными технологиями инженерии программного обеспечения и не имеющих понятия о хитростях и тонкостях программирования данной системы.
6. Данные, с которыми работает система, могут содержаться в разных файлах с несовместимыми структурами. Отсюда высокая вероятность дублирования данных, кроме того, они могут быть устаревшими, неполными либо неточными.

Организация, использующая большое количество наследуемых систем, сталкивается с серьезной проблемой. Продолжая использовать и модернизировать наследуемые системы, она тем самым значительно повышает свои расходы. Решение заменить наследуемую систему новой также связано с большими расходами, более того, новая система неспособна конкурировать с наследуемой в плане оптимальной поддержки целевой деятельности компании. Поэтому многие организации занимаются поиском новых технологий разработки ПО, позволяющих продлевать жизненный цикл наследуемых систем и снижающих затраты по их использованию. О некоторых из них читатель узнает в главах 27 и 28, посвященных вопросам эволюции программного обеспечения.

26.1. Структуры наследуемых систем

Понятие "наследуемая система" гораздо шире понятия "старые и давно используемые системы ПО", хотя именно программный компонент этих систем нас интересует больше всего. Наследуемая система представляет собой сложную социотехническую систему (см. главу 2), основанную на использовании вычислительной техники, которая включает программное обеспечение, аппаратные средства, используемые данные и бизнес-процессы. Изменения одной из составляющих системы влечет за собой изменение других ее компонентов. Эти системы разрабатывались с учетом организационных стратегий и планов конкретной организации, но не всегда учитывали объективные инженерные критерии.

Логические составляющие наследуемых систем и взаимосвязи между ними перечислены ниже и показаны на рис. 26.1.

1. *Аппаратные средства.* В большинстве своем наследуемые системы были созданы для работы на больших универсальных электронно-вычислительных машинах, которые уже не выпускаются. Эти машины отличаются дороговизной эксплуатации и несовместимы с современными вычислительными средствами.

2. *Программные средства поддержки.* Наследуемая система может основываться на самых разных средствах поддержки, начиная с операционных систем и обслуживающих программ и заканчивая компиляторами, используемыми при создании системы. Все это может быть давно устаревшим и не поддерживаться их производителями.
3. *Прикладное программное обеспечение.* Как уже отмечалось, прикладная система, обеспечивающая услуги в сфере бизнеса, обычно состоит из нескольких отдельных программ, которые разрабатывались в разное время. Часто термин "наследуемая система" относится именно к этим прикладным программам, а не ко всей системе в целом.
4. *Данные.* Это данные, с которыми работает прикладная система. Многие системы за время эксплуатации накапливают огромное количество данных, среди которых можно обнаружить как неверные, так и дубликаты, содержащиеся в разных файлах.
5. *Бизнес-процессы.* Это вид деловой активности для достижения коммерческих целей. Если взять в качестве примера страховую компанию, бизнес-процессом в ней может быть применение политики страхования, а для промышленной компании бизнес-процессом будет считаться прием заказа на производство определенного продукта и определение технологии производственного процесса.
6. *Политика и правила деловой активности.* Здесь определяются способ ведения и различные ограничения деловой активности компании. Эти политики и правила часто лежат в основе построения и эксплуатации наследуемых прикладных систем.

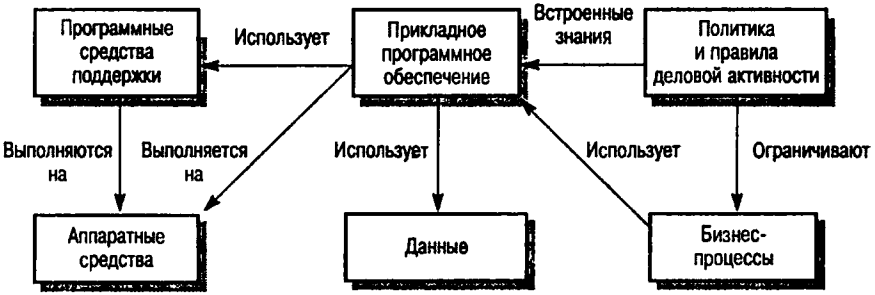


Рис. 26.1. Компоненты наследуемых систем

Другой взгляд на наследуемые системы представлен на рис. 26.2, где наследуемая система показана в виде многоуровневой модели. Каждый уровень зависит от нижнего, взаимодействуя с ним посредством интерфейса. В идеале эти интерфейсы должны позволять проводить изменения на отдельных уровнях без влияния или согласования с другими уровнями.

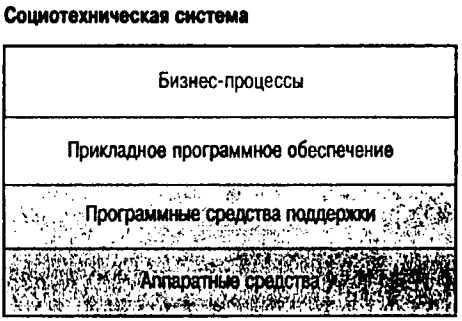


Рис. 26.2. Многоуровневая модель наследуемой системы

На практике вмешательство в одии уровень обязательно повлечет за собой изменения на других уровнях. Это происходит по нескольким причинам.

1. Изменения на каком-либо уровне в большинстве случаев связаны с внедрением новых средств. Чтобы вышестоящий уровень мог использовать эти средства, его нужно также изменить. Например, на уровень программных средств поддержки внедряется новая база данных, которая предоставляет доступ к данным с помощью Web-броузера. Тогда уровень бизнес-процессов потребуется изменить для того, чтобы иметь возможность использовать это средство.
2. Изменение программного обеспечения может снизить скорость выполнения системы, для ее повышения нужно установить новые аппаратные средства. Повышение производительности системы провоцирует внесение изменений для использования возможностей, которые ранее были недоступны.
3. Сохранение интерфейсов аппаратных средств со временем часто становится невозможным, особенно в случае кардинальных изменений в аппаратном обеспечении. Такое может случиться с компанией, которая решит перейти от мэйнфреймов (больших ЭВМ) к системе клиент/сервер (см. главу 11), где, как правило, работают с разными операционными системами. В этом случае необходима серьезная модернизация прикладного программного обеспечения.

Прикладное программное обеспечение наследуемой системы – это не только отдельная программа, а целый комплекс программ. Конечно, первоначально в системе может быть только одна программа, работающая с одним-двумя файлами данных, однако со временем в систему могут быть включены другие программы, осуществляющие обмен данными и связь с другими системными программамн. Таким же образом с поступлением новой информации пополняются изначальные файлы данных. Этот процесс отображен на рис. 26.3. Программы осуществляют обмен файлами данных, поэтому изменения в одной программе обязательно отразятся на других.

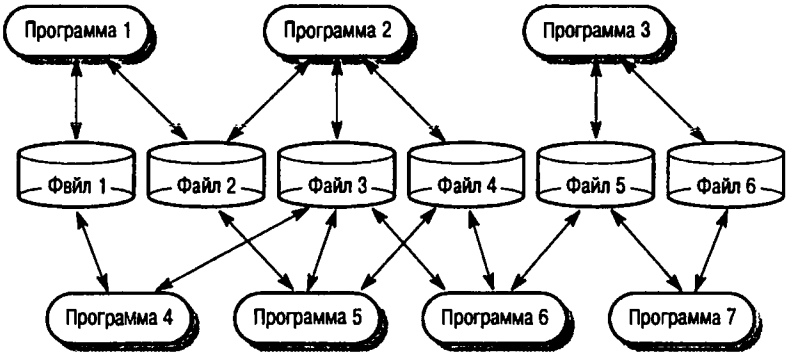


Рис. 26.3. Наследуемые прикладные системы

В наше время еще можно найти системы, которые используют отдельные файлы для хранения данных, однако большинство систем уже перешли на централизованное хранение информации в базах данных (рис. 26.4.). Преимущество такого подхода заключается в том, что для представления данных используются логические и физические модели данных. Это позволяет избежать или снизить уровень избыточности и дублирования информации, правильно оценить воздействие на данные каких-либо изменений в системе. Кро-

ме того, базы данных обеспечивают средства обработки транзакций, гарантирующие восстановление данных. Ими также обеспечивается диалоговое обновление информации.

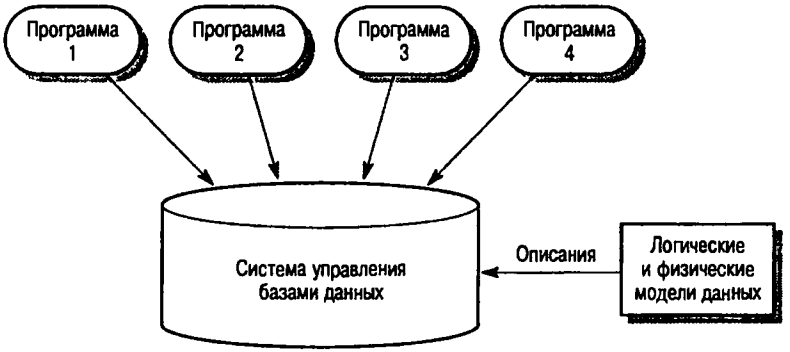


Рис. 26.4. Система с централизованной базой данных

Запросы на обновление информации могут поступать с различных терминалов и в разное время. Возьмем для примера банковскую систему, насчитывающую сотни терминалов, с которыми работают кассиры в филиалах и клиенты, пользующиеся банкоматами. Обработка всех отдельных транзакций сконцентрирована в центральной базе данных счетов. Монитор дистанционной обработки (например, система CICS от IBM) может обрабатывать и помещать в буфер вводные данные из многих различных источников. В банковской системе он принимает транзакции от филиалов и банкоматов, при этом возможна первичная обработка на месте. Затем транзакции помещаются в буфер и предоставляются в базу данных счетов в виде последовательного списка; база данных обновляет счета клиентов и подтверждает завершение обработки транзакций. Эта процедура показана на рис. 26.5.

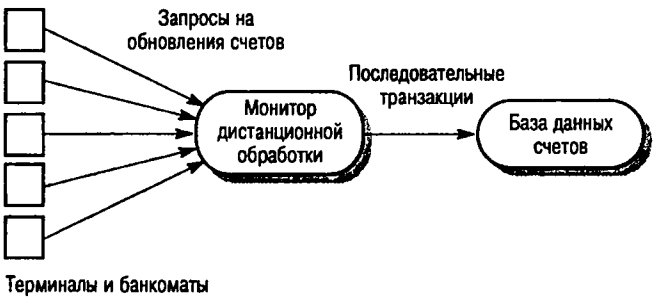


Рис. 26.5. Обработка транзакций с использованием монитора дистанционной обработки

Наследуемые системы с централизованными базами данных также имеют недостатки.

1. Система управления базой данных может быть устаревшей и несовместимой с современными СУБД, используемыми в бизнесе. Из всех современных систем баз данных, применяемых в бизнесе, наиболее эффективными считаются реляционные базы данных. Однако многие наследуемые системы используют иерархические или сетевые базы данных. Такие базы данных создавались скорее для повышения функциональности системы, чем для удобства управления данными. Современная

вычислительная техника снимает требование повышения функциональности системы, но переход к реляционным моделям данных может оказаться слишком дорогостоящим.

2. Монитор дистанционной обработки часто создавался для специализированных баз данных, рассчитанных на эксплуатацию на мэйнфреймах. Поэтому такой монитор не подойдет для современных баз данных. Эта часть системы подлежит обязательной замене, вследствие чего возрастают расходы и риск, связанные с изменениями системы.

26.2. Проектирование наследуемых систем

Практически все наследуемые системы были созданы до того, как объектно-ориентированный подход стал широко использоваться при создании ПО. Поэтому, вместо того чтобы представлять собой совокупность взаимосвязанных объектов, программы в таких системах структурированы как множество подпрограмм и функций. Каждая подпрограмма обеспечивает определенную часть функциональности системы и в случае необходимости вызывается другими подпрограммами. В некоторых же языках программирования подпрограммы оперируют собственными данными, имея в то же время доступ к совместно используемым данным. В других языках (например, ранние версии COBOL) для всех подпрограмм открыт совместный доступ к общим данным.

Стратегия функционально-ориентированного проектирования ПО предусматривает декомпозицию программ на ряд функций и подпрограмм, взаимодействующих с централизованной совместно используемой памятью (рис. 26.6.). Информация о локальном состоянии функций обрабатывается только в процессе их исполнения. Такая стратегия является частью многих структурных методов, разработанных в конце 70-х – начале 80-х годов. Она получила название "нисходящее проектирование" и "структурное проектирование" [79, 250, 345, 12*, 24*]. Сотни тысяч прикладных программ разработаны с помощью этих методов и соответствующих CASE-средств.

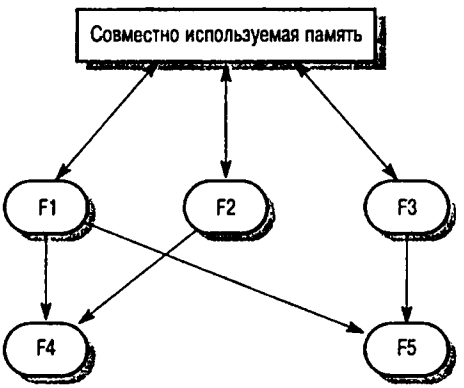


Рис. 26.6. Функционально-ориентированный подход к проектированию ПО

Функционально-ориентированное проектирование скрывает детали алгоритмов в подпрограммах и функциях, однако информация о состоянии системы при этом открыта. В этом могут таиться проблемы, поскольку функция способна изменить состояние системы непредвиденным образом. Изменения в самой функции и состоянии системы могут привести к изменениям в поведении других функций. Это большая проблема наследуемых

систем, особенно если представить, сколько разных людей вносили изменения в систему за время ее существования: один человек едва ли способен разобраться в том, каким образом взаимодействуют разные части системы.

Функциональный подход к проектированию будет эффективен лишь в том случае, если свести к минимуму количество открытой информации о состоянии системы и сделать обмен информацией более явным. Системы, которые зависят от входных данных или сигналов и не зависят от предыстории входных данных, обладают определенной функциональной направленностью. Большинство систем обработки деловой информации предназначены для обработки отдельных (дискретных) записей. Работа с новой записью не зависит от результатов обработки предыдущей. Поэтому при создании таких систем выгоднее использовать функциональное программирование.

Системы обработки деловой информации представляют собой самый большой класс наследуемых систем и разделяются на два типа.

1. *Системы пакетной обработки данных.* Ввод-вывод данных осуществляется в пакетном режиме из файлов, а не с терминала пользователя. Такими системами являются программы начисления заработной платы, выписки счетов и т.д.
2. *Системы обработки транзакций.* Ввод-вывод данных представляет собой серию транзакций, обрабатываемых системой управления базой данных, при этом транзакции генерируются терминалом пользователя.

Конечно, эти различные системы могут также использовать общие данные. Например, банк при работе со счетами использует систему обработки транзакций, но при создании выписок из банковских счетов клиентов используется система пакетной обработки данных.

Обе системы (пакетной обработки данных и обработки транзакций) действуют в соответствии с моделью "вход-процесс-выход", показанной на рис. 26.7. Системы осуществляют ввод данных из одного или нескольких источников, обрабатывают их и выдают выходные данные, которые в той или иной степени связаны с входными. В качестве примера можно рассмотреть систему телефонных счетов, где входными данными являются записи о звонках клиента и информация со счетчика коммутатора АТС, которые затем обрабатываются компьютером, в результате на выходе системы — счета за пользование телефоном.

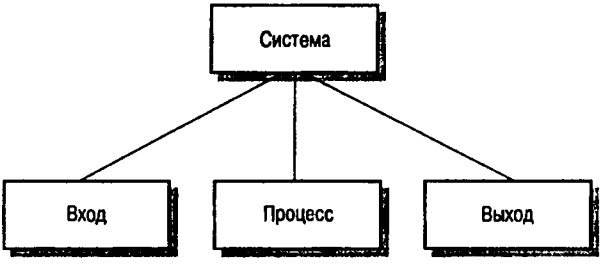


Рис. 26.7. Модель "вход-процесс-выход"

Системные компоненты ввода, обработки и вывода информации также могут быть разбиты по принципу "вход-процесс-выход", например следующим образом.

1. Компонент входа может включать непосредственный ввод информации с терминала пользователя (вход), проверку достоверности данных и исправление некоторых ошибок (процесс), затем помещение данных в очередь на обработку (выход).

2. В компонеит обработки может входить получение транзакции из очереди (вход), подсчет данных, создание новой записи по результатам подсчета (процесс) и помещение новой записи в очередь на печать (выход).
3. Компонент выхода состоит из считывания записей из очереди (вход), форматирования записей в соответствии с формой вывода (процесс) и последующей отправки их на печать (выход).

При проектировании функционально-ориентированных систем часто используются диаграммы потоков данных, которые описаны в главе 7. Диаграммы потоков данных – это функциональное представление, где прямоугольник с закругленными краями представляет функцию, выполняющую преобразование данных, а стрелка – элемент данных, обрабатываемый функцией. Файлы и другие хранилища данных представлены в виде прямоугольников. Диаграммы отображают сквозной процесс обработки, т.е. показывают все функции системы, которые взаимодействуют с данными, когда они (данные) проходят по разным стадиям обработки и преобразований.

Для иллюстрации функционально-ориентированного проектирования с помощью диаграммы потоков данных рассмотрим рис. 26.8, на котором показана структура системы по расчету заработной платы. Это система пакетной обработки данных. Она считывает информацию о служащих компании, затем рассчитываются платежи и отчисления за месяц, после чего начисляется заработная плата. Рассмотрим, как эта система реализует структуру "вход–процесс–выход".

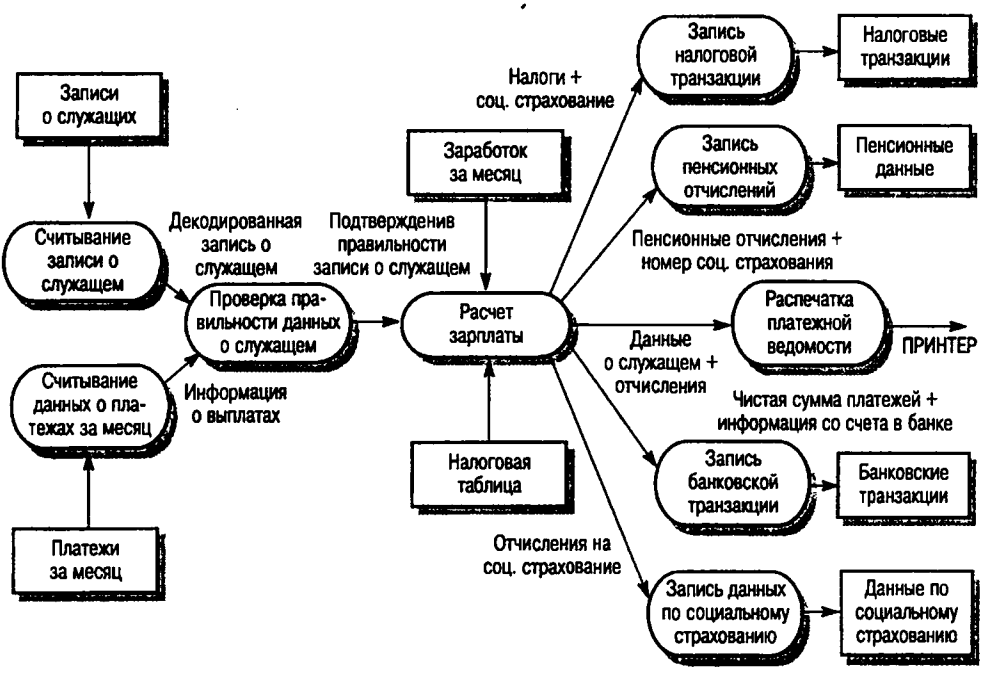


Рис. 26.8. Потоковая диаграмма системы выплаты зарплат

1. Функции "Считывание записи о служащем", "Считывание данных о платежах за месяц" и "Проверка правильности данных о служащем" реализуют ввод и проверку данных о каждом сотруднике.

2. Функция "Расчет зарплаты" обрабатывает всю информацию об окладе до удержания налогов по каждому служащему, а также данные об отчислениях с зарплаты. Результатом этого является "чистая" зарплата за месяц.
3. Функции вывода осуществляют запись в ряд файлов, в которых содержится отдельная информация по отчислениям с зарплаты. Эти файлы обрабатываются другими программами после того, как будет произведен расчет данных для всех служащих. Вслед за этим следует распечатка платежной ведомости, в которой указаны суммы выплат и отчислений.

Удачным примером системы обработки транзакций является система, контролирующая сеть банкоматов. На услуги, предоставляемые пользователям, не влияют предварительные операции, поэтому они могут рассматриваться как отдельные транзакции. В листинге 26.1 приведена упрощенная версия такой системы. Следует заметить, что вместо языка программирования Java я использовал функционально-ориентированный язык. Java – объектно-ориентированный язык программирования, поэтому не подходит для описания функционально-ориентированных систем.

Листинг 26.1. Описание системы управления банкоматами

```
ВХОД
loop
    repeat
        Печать_сообщение("Добро пожаловать! Вставьте Вашу карточку")
    until Карточка_ввод
    Счет_номер := Считывание_карточка;
    Получение_счет (PIN_код, Счет_баланс, Наличность_доступность);
ПРОЦЕСС
    if карточка_недействительна (PIN_код) then
        Задержать_карточку;
        Print ("Карточка задержана - пожалуйста, свяжитесь с Вашим
банком");
    else
    repeat
        Печать_операция_выбор_сообщение;
        Кнопка:= Получение_кнопка;
        case Получение_кнопка is
            when Наличность_только =>
                Выдача_наличность_(Наличность_доступна, Сумма_выдача);
            when Печать_остаток =>
                Печать_клиент_остаток (Счет_остаток);
            when Отчет =>
                Заказ_отчет (Счет_номер);
            when Чековая_книжка =>
                Заказ_чековая_книжка (Счет_номер);
        end case;
        Print ("Нажмите ПРОДОЛЖИТЬ для других операций либо СТОП");
        Кнопка := Получение_кнопка;
    until Кнопка = СТОП
ВЫХОД
    Извлечь_карточку;
    Print ("Пожалуйста, извлеките Вашу карточку");
    Обновление_счет (Счет_номер, Счет_выдача);
end loop;
```

В этом проекте работа системы реализуется в виде цикла, где операции начинают выполняться после ввода пользователем карточки. Системные функции (Выдача_наличность, Получение_счет, Заказ_отчет, Заказ_чековая_книжка и др.) можно определить при реализации системы. Контроль за состоянием системы, осуществляемый программой, минимален. Операции по обслуживанию клиентов выполняются индивидуально и не взаимодействуют между собой.

Очевидно, что функционально-ориентированное проектирование будет использоваться при разработке программных систем еще много лет. Конечно, эта технология не привязана к разработке наследуемых систем, она может использоваться и для создания новых систем в следующих ситуациях.

1. При создании систем обработки данных, основанных на работе с транзакциями и обновлении баз данных. Программа, обрабатывающая транзакции, не нуждается в информации о предыдущих транзакциях, поэтому нет необходимости в объектах, работающих с локальными данными. Здесь наблюдается следование модели "вход–процесс–выход", которая рассмотрена выше.
2. В компаниях, вложивших значительные средства в структурные методы, соответствующие CASE-средства и обучение персонала. Здесь будут неоправданными риск и затраты, связанные с переходом на объектно-ориентированное проектирование.

Хотя функционально-ориентированный подход во многом считается устаревшим, объектно-ориентированное проектирование в подобных ситуациях не будет оправданным. Таким образом, перед нами стоит интересная задача: обеспечить совместную работу двух подходов к программированию — объектно-ориентированного и функционально-ориентированного.

26.3. Оценивание наследуемых систем

Организации, деятельность которых во многом зависит от наследуемых систем и средства которых на их сопровождение и модернизацию ограниченны, должны хорошо подумать над тем, как получить максимум от вложений в наследуемую систему. Это прежде всего означает корректную оценку наследуемой системы и выбор наиболее подходящей стратегии ее модернизации. Существует четыре стратегических пути решения этой задачи.

1. *Полностью отказаться от системы.* Это решение применимо в случае, если система не отвечает своим задачам поддержки бизнес-процессов. Например, со времени установки системы бизнес-процессы изменились настолько, что уже практически не зависят от работы системы. Чаще всего это случается там, где универсальные ЭВМ были заменены персональными компьютерами, а устаревшее программное обеспечение было модернизировано в той мере, в которой это необходимо для продолжения его работы на ПК.
2. *Продолжить сопровождение системы.* Это решение подходит в ситуациях, когда система более или менее стабильна и все еще полезна в работе, а пользователи не требуют ее значительного изменения.
3. *Модернизировать систему для улучшения сопровождения.* Этот путь следует выбрать тогда, когда качество работы системы снизилось в результате частых изменений, причем дальнейшие изменения все еще будут необходимы.
4. *Заменить старую систему более новой.* Этот вариант применяется в том случае, если в связи с появлением современных аппаратных средств старая система становится непригодной в эксплуатации или если уже имеются подобные системы и разработка новых на их основе не будет слишком дорогостоящей.

Естественно, нет однозначного решения данной проблемы — к системе, состоящей из нескольких отдельных программ, можно применить несколько различных подходов.

При оценивании наследуемую систему нужно рассматривать под разными углами зрения [339]. С коммерческой точки зрения необходимо провести оценку полезности и пригодности системы для бизнеса. Что же касается перспективы дальнейшей работы системы, нужно в первую очередь оценить качество прикладного ПО, а также программных и аппаратных средств поддержки данной системы. Комбинация двух оценок — бизнес-пригодность и качество — поможет решить, что же делать с наследуемой системой дальше.

Для демонстрации применения такой комплексной оценки рассмотрим организацию, использующую в работе 10 наследуемых систем. Качество и бизнес-пригодность каждой из этих систем были оценены с помощью некоторых количественных показателей. Результаты оценивания представлены на рис. 26.9.

Рис. 26.9. Бизнес-пригодность и качество систем

На рис. 26.9 видно четыре группы систем, которые определяются следующими оценками.

1. *Низкое качество, низкая бизнес-пригодность.* Решение оставить такие системы в действии дорого обойдется, а отдача в бизнесе будет незначительной. Такие системы — прямые кандидаты на списание.
2. *Низкое качество, высокая бизнес-пригодность.* Такие системы вносят немалый вклад в бизнес-деятельность, поэтому списывать их нельзя. Однако низкий уровень качества означает высокие расходы на сопровождение системы. Такие системы подлежат модернизации или замене (при условии наличия другой подходящей системы).
3. *Высокое качество, низкая бизнес-пригодность.* Такие системы не очень полезны, но недороги в эксплуатации. Риск вследствие замены таких систем неоправдан, поэтому их можно оставить в работе и в дальнейшем списать.
4. *Высокое качество, высокая бизнес-пригодность.* Их можно оставить, ведь высокое качество означает отсутствие необходимости модернизации или замены системы. Поэтому с ними продолжаем работать, как и прежде.

В идеале для решения того, что делать с системой дальше, должны использоваться подобные объективные оценки. Однако неоднократно отмечалось, что в действительности при принятии таких решений главную роль играют организационные или политические соображения. Например, при слиянии компаний будут использоваться системы более

сильного (в политическом смысле) партнера, а все другие системы будут списаны. Или если руководство компании примет решение о переходе на новую вычислительную технику, то прикладные программы также подлежат замене. Также и в случае ограниченного бюджета, который не позволяет провести модернизацию системы в текущем году, поэтому будет продолжено сопровождение старой системы, даже с неблагоприятными экономическими перспективами.

26.3.1. Оценка бизнес-пригодности

Оценка бизнес-пригодности системы в определенной мере субъективна, поскольку не существует идеальных объективных методов ее определения. Здесь, как во всех случаях субъективного оценивания, полагаясь только на одно мнение, вы получите весьма искаженный результат. Чтобы избежать этого, я предлагаю подход, который предусматривает оценивание бизнес-пригодности системы под разными углами зрения. Предлагаю несколько опорных точек зрения (см. главу 6) и соответствующих вопросов, которые помогут в оценке бизнес-пригодности.

1. *Точка зрения конечного пользователя системы.* Насколько эффективна данная система для поддержки работы пользователя? Какой процент использования функциональных возможностей системы?
2. *Точка зрения заказчиков.* Является ли работа системы ясной и понятной для заказчика? Имеются ли ограничения при взаимодействии заказчика с системой?
3. *Точка зрения менеджеров.* Какое влияние оказывает система на деятельность их подразделения? Оправданы ли средства, потраченные на сопровождение системы? Насколько важными для работы подразделения являются данные, обрабатываемые системой?
4. *Точка зрения менеджеров группы сопровождения системы.* Трудно ли найти специалистов для работы с системой? Можно ли ресурсы, затрачиваемые на сопровождение системы, вложить в другие системы с большей эффективностью?
5. *Точка зрения руководства компании.* Насколько зависит от системы достижение коммерческих целей компании и решение бизнес-задач?

После определения опорных точек зрения необходимо проинтервьюировать представителей каждой группы и сравнить их ответы. Это даст четкую оценку бизнес-пригодности системы.

26.3.2. Оценка качества системы

Из представления наследуемых систем, показанного на рис. 26.2, видно, что их составными частями также являются бизнес-процесс, аппаратные средства и программные средства поддержки. Чтобы оценить качество системы, нужно изучить все ее уровни, только после этого можно поместить ее на одну из позиций в схеме, показанной на рис. 26.9. Но универсального метода оценки качества системы не существует. Оценка зависит от вида системы и деловой сферы, в которой она используется.

Оценка бизнес-процесса

Оценивание качества бизнес-процесса тесно связано с оцениванием бизнес-пригодности системы. Но учитывая, что бизнес-процесс составляет часть наследуемой системы, для решения об уровне качества необходимо иметь достаточно подробную информацию о бизнес-процессе. Это важный момент, поскольку при неудовлетворительном качестве бизнес-процесса необходимо вносить соответствующие изменения в систему.

Для оценки бизнес-процесса я бы предложил подход, ориентированный на сопоставление разных опорных точек зрения. Вот несколько вопросов, которые можно использовать для такого сопоставления.

1. Существует ли в организации четкая модель бизнес-процесса, а также процедуры для контроля соответствия бизнес-процесса этой модели?
2. Следуют ли разные подразделения компании одинаковым бизнес-процессам при выполнении одинаковых функций?
3. Каким образом специалисты, вовлеченные в процесс, адаптировали его к условиям своей работы?
4. Есть ли необходимость во взаимосвязи с другими бизнес-процессами, и насколько эта взаимосвязь ясна пользователям?
5. Поддерживается ли бизнес-процесс наследуемыми прикладными системами? Предоставляется ли необходимая информация? Требуется ли для процесса дублирование данных в разных местах?

При этом вас не должны удивлять совершенно разные ответы на эти вопросы от разных групп интервьюируемых. Например, менеджеры могут считать процесс достаточно эффективным, хотя на самом деле он таковым не является. Составляя мнение о качестве бизнес-процесса, не следует полагаться на сопроводительную документацию. Лучше сосредоточиться на ответах лиц, участвующих в реальных бизнес-процессах.

Оценка окружения

На рис. 26.2 видно, что окружение прикладной системы включает ПО поддержки (операционные системы, компиляторы, служебные программы и т.д.) и аппаратные средства, на которых исполняется система. Фактор окружения иногда является ключевым при принятии решения об изменениях в прикладной системе, поэтому важна оценка системного окружения.

Для оценивания системного окружения необходимо изучить саму систему и процесс ее сопровождения. Например, полезной окажется информация о стоимости эксплуатации аппаратных средств и ПО поддержки, количестве сбоев аппаратных средств за определенный период времени, а также о частоте настройки ПО поддержки.

В табл. 26.1 приведены факторы, которые будут полезными при оценке системного окружения. Обратите внимание, что не все факторы чисто технического характера. Например, необходимо оценивать также надежность поставщиков аппаратных средств и программного обеспечения. Если поставщики ушли с рынка, это означает отсутствие надлежащего сопровождения системы.

Таблица 26.1. Факторы системного окружения

Фактор	Вопросы для оценки фактора
Стабильность поставщиков аппаратных средств и ПО	Присутствует ли поставщик на рынке? Насколько стабилен поставщик в финансовом плане и каковы прогнозы относительно его присутствия на рынке? Если поставщик ушел с рынка, то кем осуществляется сопровождение систем?
Количество сбоев аппаратных средств и ПО	Характеризуются ли аппаратные средства высоким уровнем сбоев в работе? Является ли ПО поддержки причиной аварийных перезагрузок системы?

Окончание табл. 26.1

Фактор	Вопросы для оценки фактора
Возраст аппаратных средств и ПО	Каков "возраст" аппаратного и программного обеспечения? Даже если оно работает без сбоев, переход к новым системам может оказаться экономически выгодным
Производительность	Насколько производительность системы соответствует требованиям современных бизнес-процессов? Испытывают ли пользователи неудобства, связанные с проблемами производительности системы?
Необходимость в средствах поддержки	Какие средства поддержки требуются для сопровождение программного и аппаратного обеспечения? Если на сопровождение уходит значительное количество средств, то может разумнее заменить систему?
Стоимость эксплуатации	Насколько велика стоимость эксплуатации аппаратных средств и затраты на закупку лицензий на программные средства поддержки? Затраты на сопровождение устаревшей техники могут быть несравнимо выше, чем на содержание более современной техники. Кроме того, ежегодное обновление лицензий на ПО может также дорого обходиться
Способность к взаимодействию с другими системами	Возникают ли проблемы, связанные с интерфейсом между данной и другими системами? Можно ли использовать компиляторы или другие средства с текущей версией операционной системы? Требуется ли эмуляция аппаратных средств?

Оценка прикладного ПО

Процесс оценивания качества существующего прикладного программного обеспечения значительно отличается от процесса оценивания качества во время разработки ПО, что отмечалось в главе 24. Это обусловлено спецификой наследуемых систем, состоящей в том, что они создавались с применением тех технологий и стандартов, которые, может быть, уже отошли в прошлое, структура системы изменена вследствие частых изменений, а системная документация, возможно, устарела. Некоторые факторы, которые следует учитывать при оценке качества прикладного ПО, описаны в табл. 26.2.

Таблица 26.2. Факторы, используемые при оценке качества прикладного ПО

Фактор	Вопросы для оценки фактора
Простота понимания	Насколько трудно понять исходный код действующей системы? Каков уровень сложности используемых управляющих структур? Присвоены ли переменным значащие имена, показывающие их назначение?
Документация	Какая системная документация имеется в наличии? Является ли эта документация полной, последовательной и отвечающей современным требованиям?

Окончание табл. 26.2

Фактор	Вопросы для оценки фактора
Данные	Есть ли четкая модель данных, используемых в системе? Дублируются ли данные в разных файлах системы?
Производительность	Соответствует ли качество выполнения системы современным требованиям? Влияет ли производительность системы на работу пользователей?
Язык программирования	Есть ли современные компиляторы для языка программирования, с помощью которого создавалась система? Используется ли этот язык для создания современных систем?
Управление конфигурацией	Распространяется ли действие системы управления конфигурацией на все версии всех частей системы? Есть ли четкое описание всех версий системных компонентов?
Тестовые данные	Имеются ли тестовые данные? Есть ли записи о тестированиях, проведенных после введения в систему новых компонентов?
Обслуживающий персонал	Реально ли найти специалистов для обслуживания данной системы? Много ли профессионалов, способных разобраться с этой системой?

При проведении анализа качества системы также полезными будут количественные показатели.

1. *Количество изменений, внесенных в систему.* Частые изменения искажают структуру системы и усложняют проведение последующих модернизаций. Чем выше этот показатель, тем менее качественной будет система.
2. *Количество пользовательских интерфейсов, используемых системой.* Этот показатель особенно важен для систем, интерфейс которых основан на использовании различных форм ввода (каждую форму можно рассматривать как отдельный интерфейс). Чем больше различных интерфейсов, тем чаще будут встречаться в них несоответствия и избыточность.
3. *Объем данных, используемых в системе.* Высокие значения этого показателя (количество файлов, размер базы данных и т.д.) показывают значительную сложность системы.

Несмотря на несомненную полезность этих показателей, получение их числовых значений может потребовать значительных расходов. Кроме того, это также не абсолютные и не универсальные характеристики качества наследуемых систем. При использовании количественных показателей следует учесть еще размер и возраст системы.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Наследуемой называют систему, разработанную давно и долго эксплуатируемую, но все еще необходимую и полезную для бизнес-деятельности.

- Наследуемая система — это не только прикладные программы. Это комплексные социотехнические компьютерные системы, составными частями которых являются бизнес-процессы, прикладное программное обеспечение, ПО поддержки и аппаратные средства.
- Большинство наследуемых систем созданы с использованием функционально-ориентированного проектирования, вследствие чего они представляют собой комплекс взаимодействующих подпрограмм и функций, взаимосвязанных посредством передаваемых параметров и совместно используемых данных.
- Наследуемые системы, работающие в деловой сфере, подразделяются на два типа: те, которые работают с транзакциями, и те, которые обрабатывают пакеты данных. Оба типа систем соответствуют структурной модели "вход–процесс–выход".
- Решение о том, что следует делать с системой (заменить, модернизировать либо оставить в использовании), основывается на оценках бизнес-пригодности, качества прикладного ПО и системного окружения.
- Бизнес-пригодность системы определяется ее возможностью содействовать выполнению бизнес-задач.
- Качество системы зависимо от качества бизнес-процесса, качества прикладного ПО, а также от качества аппаратных средств и программных средств поддержки.

Упражнения

26.1. Объясните, чем определяется важность наследуемых систем в деловой сфере.

26.2. Приведите три причины усложнения понимания системы вследствие участия многих специалистов в изменениях системы.

26.3. Какие проблемы могут возникнуть в случае программирования разных частей системы на различных языках?

26.4. Какова роль монитора дистанционной обработки при сборе системой данных от разных терминалов? Каким образом современные системы типа клиент/сервер снижают нагрузку на монитор дистанционной обработки?

26.5. Большинство наследуемых систем созданы на основе функционально-ориентированного подхода. Объясните, почему функционально-ориентированное проектирование наследуемых систем эффективнее объектно-ориентированного.

26.6. С учетом представленной ниже информации расширьте систему расчета заработной платы (см. рис. 26.8) и постройте соответствующую диаграмму потока данных, описывающую вычисления в этой системе.

- Учетная запись о работнике содержит тарифный разряд, определяющий размер его заработной платы.
- Сверхурочная работа, если выплаты за нее ниже определенной суммы, оплачивается на уровне основной почасовой ставки. Тариф сверхурочных часов указан в учетных документах работника.
- Сумма удержанных налогов зависит от специального налогового кода работника, который указан в его учетной записи, а также от оклада. Суммы окладов и ежемесячных отчислений, зависящих от налогового кода, указаны в налоговой таблице.

26.7. Чем можно обосновать списание системы даже тогда, когда она имеет высокие оценки качества и бизнес-пригодности?

26.8. Предложите 10 вопросов, которые можно задать пользователям системы для оценки бизнес-процесса.

26.9. Объясните, почему проблемы с программным обеспечением поддержки могут стать причиной замены наследуемых систем.

26.10. Руководство компании поручило вам организовать и провести оценку системы таким образом, чтобы доказать необходимость замены системы, поскольку она устарела и непригодна к использованию. Вы знаете, что это приведет к сокращению ряда специалистов, обслуживающих старую систему. Ваша оценка доказывает обратное — система отличается высоким уровнем качества и бизнес-пригодности. Как вы отобразите этот факт в отчете руководству компании?

27

Модернизация программного обеспечения

Цели

Настоящая глава посвящена вопросам изменения программного обеспечения и описывает различные способы модификации программных систем. Прочитав эту главу, вы должны:

- знать основные стратегии модернизации программных систем, а именно: сопровождение системы, эволюцию системной архитектуры и реинжиниринг программного обеспечения;
- ориентироваться в принципах сопровождения ПО и понимать причины увеличения расходов на сопровождение систем;
- знать, каким образом наследуемую систему можно преобразовать в распределенную систему клиент/сервер, чтобы продлить срок эксплуатации системы и обеспечить более эффективное ее использование на основе современных аппаратных средств.

Содержание

Невозможно создать систему, которая не потребует изменений в будущем. Как только программное обеспечение вводится в эксплуатацию, возникают новые требования к системе, обусловленные непрерывным развитием бизнес-процессов и все возрастающими общими требованиями к программным системам. Иногда в системе следует изменить некоторые составляющие в целях повышения производительности или улучшения других характеристик, а также для исправления обнаруженных ошибок. Все это требует дальнейшего развития системы после ее ввода в эксплуатацию.

Полная зависимость организаций от программного обеспечения, которое к тому же обходится в достаточно круглую сумму, объясняет исключительную важность серьезного отношения к ПО. Это предусматривает дополнительные вложения в эволюцию уже эксплуатируемой системы с тем, чтобы обеспечить прежний уровень ее производительности.

Существует несколько стратегических подходов к процессу модернизации ПО [339].

1. *Сопровождение программного обеспечения.* Это наиболее часто используемый подход, который заключается в изменении отдельных частей ПО в ответ на растущие требования, но с сохранением основной системной структуры. Подробнее этот вопрос освещен в разделе 27.2.
2. *Эволюция системной архитектуры.* Этот подход более радикальный, чем сопровождение ПО, так как предполагает существенные изменения в программной системе. Эта стратегия модернизации ПО подробно раскрыта в разделе 27.3.
3. *Реинжиниринг программного обеспечения.* Кардинально отличается от других подходов, так как модернизация предусматривает не внесение каких-то новых компонентов, а наоборот, упрощение системы и удаление из нее всего лишнего. При этом возможны изменения в архитектуре, но без серьезных переделок. Этому вопросу посвящена глава 28.

Приведенные стратегии не исключают одна другую. Иногда для упрощения системы перед изменением архитектуры или для переделки некоторых ее компонентов применяется реинжиниринг. Некоторые части системы заменяются серийными, а более стабильные системные компоненты продолжают функционирование. Как уже упоминалось в главе 26, выбор стратегии модернизации системы основывается не только на технических характеристиках, но и на том, насколько хорошо система поддерживает деловую активность компании.

Разные стратегии могут также применяться к отдельным частям системы или к отдельным программам наследуемой системы. Сопровождение приемлемо для программ со стабильной и четкой структурой, не требующей особого внимания. Для других программ, которые постоянно контактируют со многими пользователями, можно изменить архитектуру так, чтобы интерфейс пользователя запускался на машине клиента. Еще один компонент в этой же системе можно заменить аналогичной программой стороннего производителя. Однако при реинжиниринге обычно необходимо изменять все компоненты системы.

Изменения в ПО служат причиной появления многочисленных версий системы и ее компонентов. Поэтому особенно важно внимательно следить за всеми этими изменениями, а также за тем, чтобы версия компонента соответствовала той версии системы, в которой он применяется. Управление изменениями системы называется управлением конфигурацией и обсуждается в главе 29.

27.1. Динамика развития программ

Под динамикой развития программ подразумевается исследование изменений в программной системе. Основной работой в этой области является [213]. Результатом этих исследований стало появление ряда "законов" Лемана (Lehman), относящихся к модернизации систем. Считается, что эти законы неизменны и применимы практически во всех случаях. Они сформулированы после исследования процесса создания и эволюции ряда больших программных систем. Эти законы (в сущности, не законы, а гипотезы) приведены в табл. 27.1.

Таблица 27.1. Законы Лемана

Закон	Описание
Непрерывность модернизации	Для программ, эксплуатируемых в реальных условиях, модернизация — это необходимость, иначе их полезность снижается
Возрастающая сложность	По мере развития программы становятся все более сложными. Для упрощения или сохранения их структуры необходимы дополнительные затраты
Эволюция больших систем	Процесс развития систем саморегулируемый. Такие характеристики системы, как размер, время между выпусками очередных версий и количество регистрируемых ошибок, для каждой версии программы остаются практически неизменными
Организационная стабильность	Жизненный цикл системы относительно стабилен, независимо от средств, выделяемых (или не выделяемых) на ее развитие
Стабильность количества изменений	За весь жизненный цикл системы количество изменений в каждой версии остается приблизительно одинаковым

Из первого закоиа вытекает необходимость постоянного сопровождения системы. При изменении окружения, в котором работает система, появляются новые требования, и система должна неизбежно изменяться с тем, чтобы им соответствовать. Изменения системы носят циклический характер, когда новые требования порождают появление новой версии системы, что, в свою очередь, вызывает изменения системного окружения; это находит отражение в формировании новых требований к системе и т.д.

Второй закон констатирует нарушение структуры системы после каждой модификации. Это в полной мере демонстрируют наследуемые системы, которые рассматриваются в главе 26. Единственным способом избежать этого, по всей видимости, является только профилактическое обслуживание, которое, однако, требует средств и времени. При этом совершенствуется структура программы без изменения ее функциональности. Поэтому в бюджете, предусмотренном на содержание системы, следует также учесть и эти дополнительные затраты.

Самым спорным и, пожалуй, самым интересным законом Лемана является третий. Согласно этому закону, все большие системы имеют собственную динамику изменений, которая устанавливается на начальном этапе разработки системы. Этим определяются возможности сопровождения системы и ограничивается количество модификаций. Предполагается, что этот закон является результатом действия фундаментальных структурных и организационных факторов. Как только система превышает определенный размер, она начинает действовать подобно некой инерционной массе. Размер становится препятствием для новых изменений, поскольку эти изменения с большой вероятностью станут причиной ошибок в системе, которые снизят эффективность нововведений в новой версии системы.

Четвертый закон Лемана утверждает, что крупные проекты по разработке программного обеспечения действуют в режиме "насыщения". Это означает, что изменения ресурсов или персонала оказывает незначительное влияние на долгосрочное развитие системы. Это, правда, уже указано в третьем законе, который утверждает, что развитие программы не зависит от решений менеджмента. Этим законом также утверждается, что крупные команды программистов неэффективны, так как время, потраченное на общение и внутрикомандные связи, превышает время непосредственной работы над системой.

Пятый закон затрагивает проблему увеличения количества изменений с каждой новой версией программы. Расширение функциональных возможностей системы каждый раз сопровождается новыми ошибками в системе. Таким образом, масштабное расширение функциональных возможностей в одной версии означает необходимость последующих доработок и исправления ошибок. Поэтому в следующей версии уже будут проведены незначительные модификации. Таким образом, менеджер, формируя бюджет для внесения крупных изменений в версию системы, не должен забывать о необходимости разработки следующей версии с исправленными ошибками предыдущей версии.

В основном законы Лемана выглядят весьма разумными и убедительными. При планировании сопровождения они обязательно должны учитываться. Случается, что по коммерческим соображениям законами следует пренебречь. Например, это может быть обусловлено маркетингом, если существует необходимость провести ряд модификаций системы в одной версии. В результате все равно получится так, что одна или несколько следующих версий будут связаны с исправлением ошибок.

Может показаться, что большие различия между последовательными версиями одной и той же программы опровергнут законы Лемана. Например, Microsoft Word превратилась из простой программы текстовой обработки, требующей 25 Кбайт памяти, в огромную систему с множеством функций. Теперь, для того чтобы работать с этой программой, нужно много памяти и быстродействующий процессор. Эволюция этой программы противоречит четвертому и пятому законам Лемана. Однако я подозреваю, что это все-таки не одна и та же программа, которая просто подверглась ряду изменений. Думаю, программа была существенно переработана; по сути, была разработана новая программа, но в рекламных целях был сохранен единый логотип.

27.2. Сопровождение программного обеспечения

Сопровождение — это обычный процесс изменения системы после ее поставки заказчику. Эти изменения могут быть как элементарно простыми (исправление ошибок программирования), так и более серьезными, связанными с корректировкой отдельных недоработок либо приведением в соответствие с новыми требованиями. Как упоминалось в вводной части главы, сопровождение не связано со значительным изменением архитектуры системы. При сопровождении тактика простая: изменение существующих компонентов системы либо добавление новых.

Существует три вида сопровождения системы.

1. *Сопровождение с целью исправления ошибок.* Обычно ошибки в программировании достаточно легко устранимы, однако ошибки проектирования стоят дорого и требуют корректировки или перепрограммирования некоторых компонентов. Самые дорогие исправления связаны с ошибками в системных требованиях, так как здесь может понадобиться перепроектирование системы.

2. *Сопровождение с целью адаптации ПО к специфическим условиям эксплуатации.* Это может потребоваться при изменении определенных составляющих рабочего окружения системы, например аппаратных средств, операционной системы или программных средств поддержки. Чтобы адаптироваться к этим изменениям, система должна быть подвергнута определенным модификациям.
3. *Сопровождение с целью изменения функциональных возможностей системы.* В ответ на организационные или деловые изменения в организации могут измениться требования к программным средствам. В таких случаях применяется данный тип сопровождения. Наиболее существенные изменения при этом претерпевает именно программное обеспечение.

На практике однозначно четкое разграничение между различными видами сопровождения провести достаточно сложно. Ошибки в системе могут быть выявлены в том случае, если, например, система использовалась непредсказуемым способом. Поэтому наилучший способ исправления ошибок – расширение функциональных возможностей программы с тем, чтобы сделать работу с ней как можно проще. При адаптации программного обеспечения к новому рабочему окружению расширение функциональных возможностей системы будет способствовать улучшению ее работы. Также добавление определенных функций в программу может оказаться полезным, если в случае ошибок был изменен шаблон использования системы и побочным действием при расширении функциональных возможностей будет удаление ошибок.

Перечисленные типы сопровождения широко используются, хотя им подчас дают разные названия. Сопровождение с целью исправления ошибок обычно называют корректирующим. Название "адаптивное сопровождение" может относиться как к адаптации к новому рабочему окружению, так и к новым требованиям. Усовершенствование программного обеспечения может означать улучшение путем соответствия новым требованиям, а также усовершенствование структуры и производительности с сохранением функциональных возможностей. Я намеренно не употребляю здесь все эти названия, чтобы избежать излишней путаницы.

Рис. 27.1. Распределение типов сопровождения

Найти современные данные относительно того, как часто используется тот или иной тип сопровождения, будет нелегко. Согласно исследованиям [218], которые уже несколько устарели, 65% сопровождения связано с выполнением новых требований, 18% отводится на изменения системы с целью адаптации к новому окружению и 17% связано с исправлением ошибок (рис. 27.1.). Десятилетие спустя в работе [259] определены похожие соотношения.

Из этого можно определить, что исправление ошибок не является самым распространенным видом сопровождения. Модернизация системы в соответствии с новым рабочим окружением либо в соответствии с новыми требованиями более эффективна. Поэтому сопровождение само по себе является естественным процессом продолжения разработки системы со своими процессами проектирования, реализации и тестирования. Таким образом, спиральная модель, показанная на рис. 27.2, лучше представляет процесс развития ПО, чем каскадная модель (см. рис. 3.1.), где сопровождение рассматривается как отдельный процесс.

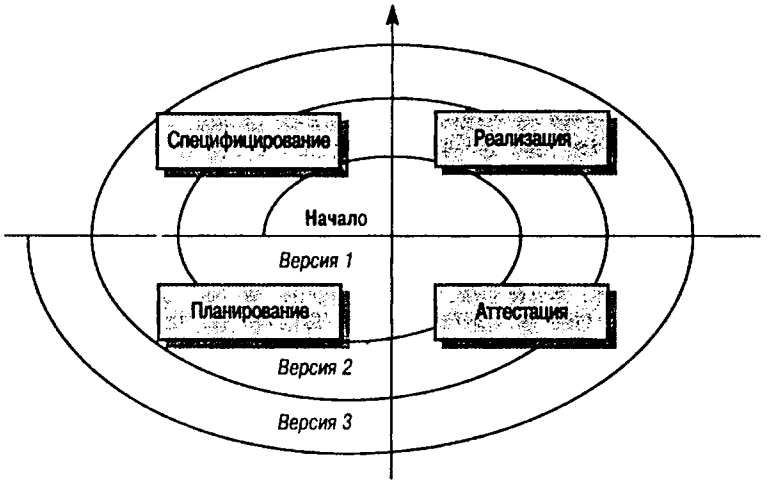


Рис. 27.2. Спиральная модель развития ПО

Значительная часть бюджета большинства организаций уходит на сопровождение ПО, а не на само использование программных систем. В 1980-х годах было обнаружено [218], что во многих организациях по меньшей мере 50% всех средств, потраченных на программирование, идет на развитие уже существующих систем. В работе [235] определено похожее соотношение затрат на различные виды сопровождения, при этом от 65 до 75% средств общего бюджета расходуется на сопровождение. Так как предприятия заменяют старые системы коммерческим ПО, например программами планирования ресурсов, эти цифры никак не будут уменьшаться. Поэтому можно утверждать, что изменение ПО все еще остается доминирующим в статье затрат организаций на программное обеспечение.

Соотношение между величинами средств на сопровождение и на разработку может быть разным в зависимости от предметной области, где эксплуатируется система. Для прикладных систем, работающих в деловой сфере, соотношение затрат на сопровождение в основном сравнимо со средствами, потраченными на разработку. Для встроенных систем реального времени затраты на сопровождение могут в четыре раза превышать стоимость самой разработки. Высокие требования в отношении производительности и

надежности таких систем предполагают их жесткую структуру, которая труднее поддается модификации.

Можно получить значительную общую экономию средств, если заранее потратить финансы и усилия на создание системы, не требующей дорогостоящего сопровождения. Весьма затратно проводить изменения в системе после ее поставки заказчику, поскольку для этого требуется хорошо знать систему и провести анализ реализации этих изменений. Поэтому усилия, потраченные во время разработки программы на снижение стоимости такого анализа, автоматически снизят и затраты на сопровождение. Такие технологии разработки ПО, как формирование четких требований, объектно-ориентированное программирование и управление конфигурацией, способствуют снижению стоимости сопровождения.

На рис. 27.3 показано, как снижается стоимость сопровождения хорошо разработанных систем. Здесь при разработке системы 1 выделено дополнительно $25 000 для облегчения процесса сопровождения. В результате за время эксплуатации системы это помогло сэкономить около $100 000. Из сказанного следует, что увеличение средств на разработку системы пропорционально снизит затраты на ее сопровождение.

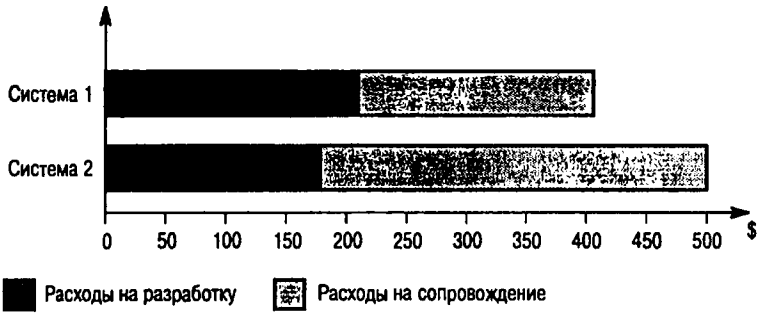


Рис. 27.3. Расходы на разработку и сопровождение систем

Причиной высоких затрат на сопровождение является сложность модернизации системы после ее внедрения, поскольку расширить функциональные возможности намного легче в процессе создания системы. Ниже приведены ключевые факторы, которые определяют стоимость разработки и сопровождения и могут привести к подорожанию сопровождения.

1. *Стабильность команды разработчиков.* Вполне естественно, что после внедрения системы команда разработчиков распадается, специалисты будут работать над другими проектами. Новым членам команды или же отдельным специалистам, которые возьмут на себя дальнейшее сопровождение системы, будет трудно понять все ее особенности. Поэтому на понимание системы перед внесением в нее изменений уходит много времени и средств.
2. *Ответственность согласно контракту.* Контракт на сопровождение обычно заключается отдельно от договора на разработку программы. Более того, часто контракт на сопровождение может получить фирма, сама не занимающаяся разработкой. Вместе с фактором нестабильности команды это может стать причиной отсутствия в команде стимула создать легкоизменяемую, удобную в сопровождении систему. Если членам команды выгодно пойти кратчайшим путем с минимальными затратами усилий, то вряд ли они откажутся от этого даже с риском повышения последующих затрат на сопровождение.

3. *Квалификация специалистов.* Специалисты, занимающиеся сопровождением, часто не знакомы с предметной областью, где эксплуатируется система. Сопровождение не пользуется популярностью среди разработчиков. Это считается менее квалифицированной разработкой и часто поручается младшему персоналу. Более того, старые системы могут быть написаны на устаревших языках программирования, не знакомых молодым специалистам и требующих дополнительного изучения.
4. *Возраст и структура программы.* С возрастом структура программ нарушается вследствие частых изменений, поэтому их становится сложнее понимать и изменять. Кроме того, многие наследуемые системы были созданы без использования современных технологий. Они никогда не отличались хорошей качественной структурой; изменения, сделанные в них, были направлены скорее на повышение эффективности функционирования, чем на повышение удобства сопровождения. Документация на старые системы часто бывает неполной либо вообще отсутствует.

Первые три проблемы объясняются тем, что многие организации все еще делают различие между разработкой системы и ее сопровождением. Сопровождение считается делом второстепенным, поэтому нет никакого желания инвестировать средства для снижения затрат на будущее сопровождение. Руководству организаций необходимо помнить, что у систем редко бывает четко определенный срок функционирования, наоборот, они могут находиться в эксплуатации в той либо иной форме неограниченное время.

Дилемма заключается в следующем: или создавать системы и поддерживать их до тех пор, пока это возможно, и затем заменять их новыми, или разрабатывать постоянно эволюционирующие системы, которые могут изменяться в соответствии с новыми требованиями. Их можно создавать на основе наследуемых систем, улучшая структуру последних с помощью реинжиниринга (см. главу 28), либо путем изменения архитектуры этих систем, что подробно рассматривается в разделе 27.3.

Последняя проблема, а именно нарушение структуры, является самой простой из них. Технология реинжиниринга поможет усовершенствовать структуру и повысить понимаемость системы. В подходящих случаях адаптировать систему к новым аппаратным средствам может и преобразование архитектуры (см. далее). Профилактические меры при сопровождении будут полезны, если возникнет необходимость усовершенствовать систему и сделать ее более удобной для изменений.

27.2.1. Процесс сопровождения

Процессы сопровождения могут быть самыми разными, что зависит от типа программного обеспечения, технологии его разработки, а также от специалистов, которые непосредственно занимались созданием системы. Во многих организациях сопровождение носит неформальный характер. В большинстве случаев разработчики получают информацию о проблемах системы от самих пользователей в устной форме. Другие же компании имеют формальный процесс сопровождения со структурированной документацией на каждый его этап. Но на самом общем уровне любые процессы сопровождения имеют общие этапы, а именно: анализ изменений, планирование версий, реализация новой версии системы и поставка системы заказчику.

Процесс сопровождения начинается при наличии достаточного количества запросов на изменения от пользователей, менеджеров или покупателей. Далее оцениваются возможные изменения с тем, чтобы определить уровень модернизации системы, а так-

же стоимость внедрения этих изменений. Если принимается решение о модернизации системы, начинается этап планирования новой версии системы. Во время планирования анализируется возможность реализации всех необходимых изменений, будь то исправление ошибок, адаптация или расширение функциональных возможностей системы. Только после этого принимается окончательное решение о том, какие именно изменения будут внесены в систему. Когда изменения реализованы, выходит очередная версия системы. На рис. 27.4, заимствованном из книги [13], представлен весь процесс модернизации.

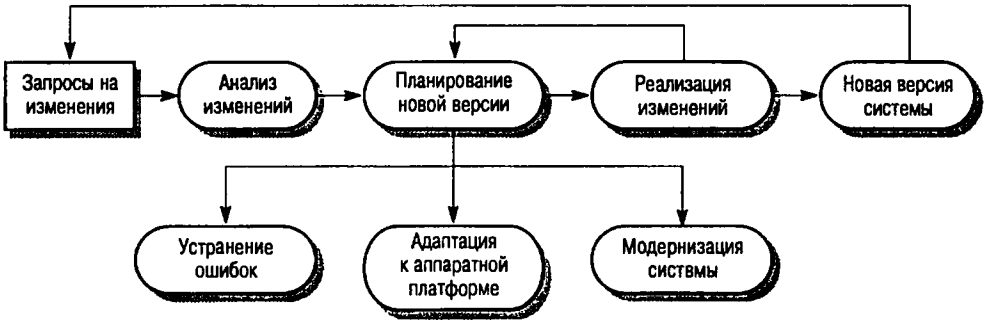


Рис. 27.4. Схема процесса модернизации

В идеале процесс модернизации должен привести к изменению системной спецификации, архитектуры и программной реализации (рис. 27.5). Новые требования должны отражать изменения, вносимые в систему. Ввод в систему новых компонентов требует ее перепроектирования, после чего необходимо повторное тестирование системы. Для анализа вносимых изменений при необходимости можно создать прототип системы. На этой стадии проводится подробный анализ изменений, при котором могут выявиться те последствия модернизации, которые не были замечены при начальном анализе изменений.

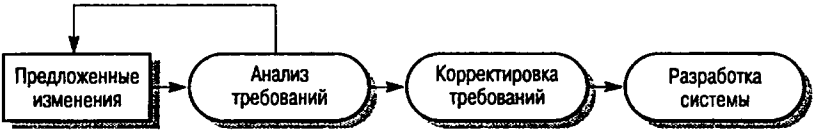


Рис. 27.5. Выполнение модернизации

Иногда в экстренных случаях требуется быстрое внесение изменений, например по следующим причинам.

- Сбой в системе, вследствие чего возникла чрезвычайная ситуация, требующая экстренного вмешательства для продолжения нормальной работы системы.
- Изменение рабочего окружения системы с непредусмотренным влиянием на систему.
- Неожиданные изменения в деловой сфере организации (из-за действий конкурентов или из-за введения нового законодательства).

В таких случаях быстрая реализация изменений имеет большую важность, чем четкое следование формальностям процесса модернизации системы. Вместо того чтобы изменять требования или структуру системы, лучше быстро внести коррективы в программный код (рис. 27.6). Однако этот подход опасен тем, что требования, системная архитек-

тура и программный код постепенно теряют целостность. Этого трудно избежать, если принять во внимание необходимость быстрого выполнения задания, когда тщательная доработка системы откладывается на потом. Если разработчик, который изменил код, внезапно уходит из команды, то его коллеге будет сложно привести в соответствие сделанным изменениям спецификацию и структуру системы.

Рис. 27.6. Процесс экстренной модернизации системы

Еще одна проблема срочных изменений системы состоит в том, что из-за дефицита времени из двух возможных решений будет принято не самое лучшее (в аспекте сохранения структуры системы), а то, которое можно быстрее и эффективнее реализовать. В идеале после срочной коррекции кода системы запрос на изменения должен все еще оставаться в силе. Поэтому после тщательного анализа изменений можно отменить внесенные изменения и принять более оптимальное решение для модернизации системы. Однако на практике такая возможность используется крайне редко, чаще всего в силу субъективных причин.

27.2.2. Прогнозирование сопровождения

Менеджеры терпеть не могут сюрпризов, особенно если они выливаются в непредсказуемо высокие затраты. Поэтому лучше предусмотреть заранее, какие изменения возможны в системе, с какими компонентами системы будет больше всего проблем при сопровождении, а также рассчитать общие затраты на сопровождение в течение определенного периода времени. На рис. 27.7 представлены различные типы прогнозов, связанные с сопровождением, и показано, на какие вопросы они должны ответить.

Прогнозирование количества запросов на изменения системы зависит от понимания взаимосвязей между системой и ее окружением. Некоторые системы находятся в достаточно сложной взаимозависимости с внешним окружением и изменение окружения обязательно повлияет на систему. Для того чтобы правильно судить об этих взаимоотношениях, необходимо оценить следующие показатели.

1. *Количество и сложность системных интерфейсов.* Чем больше системных интерфейсов и чем более сложными они являются, тем выше вероятность изменений в будущем.
2. *Количество изменяемых системных требований.* Как упоминалось в главе 6, требования, отражающие деловую сферу или стандарты организации, чаще изменяются, чем требования, описывающие предметную область.
3. *Бизнес-процессы, в которых используется данная система.* По мере развития бизнес-процессы приводят к появлению новых требований к системе.

Чтобы корректно спрогнозировать процесс сопровождения, нужно знать количество и типы взаимосвязей между разными компонентами системы, а также учитывать сложность этих компонентов. Различные виды сложности систем изучались в работах [151, 232]. Другие исследования посвящены взаимосвязям между сложностью систем и процессом сопровождения [18, 195]. Все эти исследования показали, что, чем выше сложность системы и ее компонентов, тем более дорогостоящим окажется сопровождение, чего и следовало ожидать.

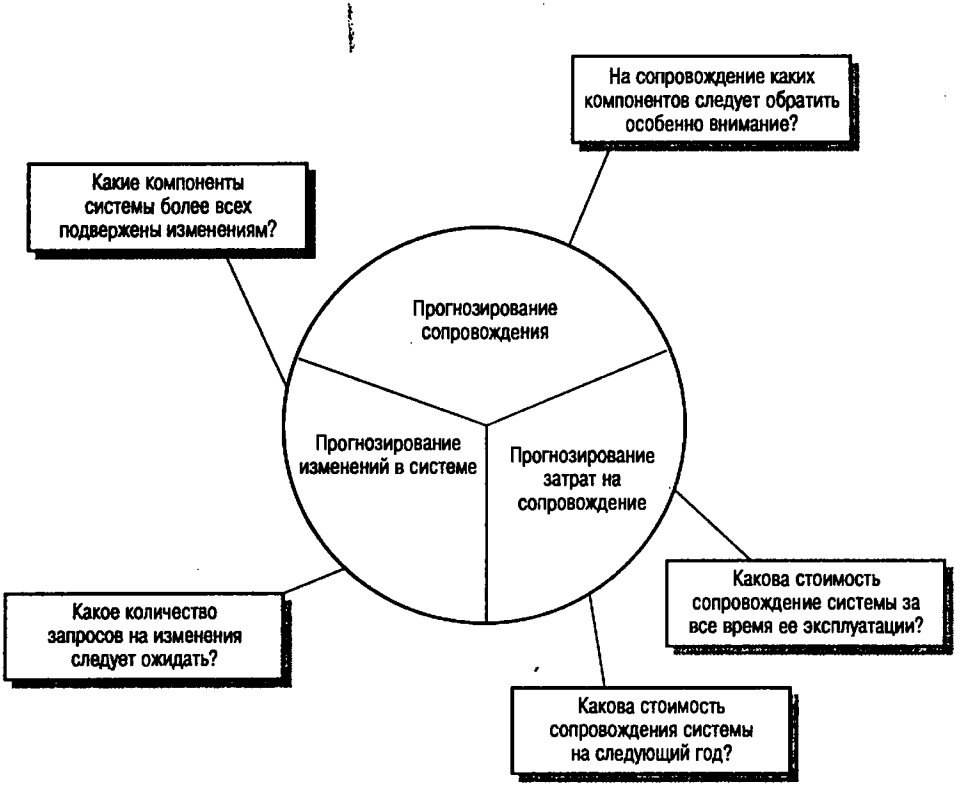


Рис. 27.7. Прогнозирование сопровождения

Прогнозирование количества запросов на изменения системы зависит от понимания взаимосвязей между системой и ее окружением. Некоторые системы находятся в достаточно сложной взаимозависимости с внешним окружением и изменение окружения обязательно повлияет на систему. Для того чтобы правильно судить об этих взаимоотношениях, необходимо оценить следующие показатели.

1. *Количество и сложность системных интерфейсов.* Чем больше системных интерфейсов и чем более сложными они являются, тем выше вероятность изменений в будущем.
2. *Количество изменяемых системных требований.* Как упоминалось в главе 6, требования, отражающие деловую сферу или стандарты организации, чаще изменяются, чем требования, описывающие предметную область.
3. *Бизнес-процессы, в которых используется данная система.* По мере развития бизнес-процессы приводят к появлению новых требований к системе.

Чтобы корректно спрогнозировать процесс сопровождения, нужно знать количество и типы взаимосвязей между разными компонентами системы, а также учитывать сложность этих компонентов. Различные виды сложности систем изучались в работах [151, 232]. Другие исследования посвящены взаимосвязям между сложностью систем и процессом сопровождения [18, 195]. Все эти исследования показали, что, чем выше сложность системы и ее компонентов, тем более дорогостоящим окажется сопровождение, чего и следовало ожидать.

В работе [18] проведено исследование ряда коммерческих программ, написанных на языке COBOL, с использованием разных методик измерения сложности, включая размер

процедур, размер модулей и количество ветвлений, что определяет сложность системы. Сравнивая сложность отдельных системных компонентов с отчетами по сопровождению, исследователи обнаружили, что снижение сложности программирования значительно сокращает расходы на сопровождение системы.

Измерение уровня сложности систем оказалось весьма полезным для выявления тех компонентов систем, которые будут особенно сложны для сопровождения. Результаты анализа ряда системных компонентов [195] показали, что сопровождение часто сосредоточено на обслуживании небольшого количества частей системы, которые отличаются особой сложностью. Поэтому экономически выгодно заменить сложные системные компоненты более простыми их версиями.

После введения системы в эксплуатацию появляются данные, позволяющие прогнозировать дальнейшее сопровождение системы. Перечисленные ниже показатели полезны для оценивания удобства сопровождения.

1. *Количество запросов на корректировку системы.* Возрастание количества отчетов о сбоях в системе означает увеличение количества ошибок, подлежащих исправлению при сопровождении. Это говорит об ухудшении удобства сопровождения.
2. *Среднее время, потраченное на анализ причин системных сбоев и отказов.* Этот показатель пропорционален количеству системных компонентов, в которые требуется внести изменения. Если этот показатель возрастает, система требует многочисленных изменений.
3. *Среднее время, необходимое на реализацию изменений.* Не следует путать этот показатель с предыдущим, хотя они тесно связаны. Здесь учитывается не время анализа системы по выявлению причин сбоев, а время реализации изменений и их документирования, которое зависит от сложности программного кода. Увеличение этого показателя означает сложность сопровождения.
4. *Количество незавершенных запросов на изменения.* С возрастанием количества таких запросов затрудняется сопровождение системы.

Для определения стоимости сопровождения используется предварительная информация о запросах на изменения и прогнозирование относительно удобства сопровождения системы. В решении этого вопроса большинству менеджеров поможет также интуиция и опыт. В модели определения стоимости COCOMO 2, описанной в главе 23, предполагается, что для оценки стоимости сопровождения понадобятся сведения об усилиях, потраченных на понимание существующего кода системы и на разработку нового.

27.3. Эволюция системной архитектуры

В процессе сопровождения большинство изменений проводится локализовано и не влияет на архитектуру системы. Однако начиная с 1980-х годов экономические показатели компьютерных систем изменилась настолько, что стало более выгодно применять распределенные, а не централизованные, как раньше, системы. Поэтому многие компании поставлены перед необходимостью преобразовать свои централизованные системы, реализованные на мэйнфреймах, в распределенные системы типа клиент/сервер (системы клиент/сервер описаны в главе 11).

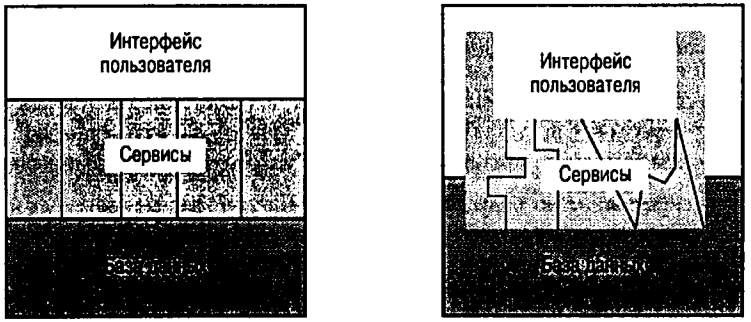
Перечислим основные причины перехода от централизованных к распределенным системам.

1. *Стоимость аппаратных средств.* Закупка и сопровождение распределенных систем клиент/сервер обойдется гораздо дешевле, чем покупка мэйнфрейма эквивалентной мощности.
2. *Усовершенствование пользовательских интерфейсов.* Многие из наследуемых систем, основанных на мэйнфремах, имеют текстовые интерфейсы, основанные на формах. Сегодня пользователям привычнее графические интерфейсы и более простое взаимодействие с системами. Такого рода интерфейсы требуют большего количества локальных вычислений и более эффективно работают в системах типа клиент/сервер.
3. *Распределенный доступ к системам.* Сейчас все больше компаний стараются децентрализовать рабочие места, что требует децентрализации компьютерных систем. При этом необходимо, чтобы компьютерные системы были доступны из разных мест и с разного оборудования. Например, сотрудники могут получить доступ к системам из собственного дома, и такую практику нужно поддерживать.

С переходом на распределенную архитектуру компьютерных систем организации значительно снижают расходы на аппаратное обеспечение и способны создать систему с более удобным интерфейсом и более современным дизайном, а также могут поддерживать практику распределенной работы. В этом переходном периоде неизбежно должно произойти преобразование системы к объектно-ориентированной модели, что, в свою очередь, может снизить затраты на сопровождение системы в будущем.

Однако нужно отметить, что преобразование архитектуры наследуемой системы является сложной задачей и требует больших расходов. Прежде чем приступить к этому процессу, необходимо провести тщательный анализ наследуемой системы, чтобы оценить реальную пользу от преобразования системной архитектуры.

Основная трудность при децентрализации наследуемых систем заключается в том, что в их структуре не существует четкого разграничения между архитектурными компонентами. Идеальной (и желательной) считается структура системы, показанная на рис. 27.8. В этом случае есть четкое разделение интерфейса пользователя, сервисов, предоставляемых системой, и базы данных. При этом сервисы четко различимы. В системе с такой структурой легко определить распределяемые компоненты, которые можно перепрограммировать и запустить на машинах клиентов.



Модель системы, идеальной для децентрализации

Реальные наследуемые системы

Рис. 27.8. Идеальная и реальная структуры систем

Но реальные наследуемые системы напоминают систему, представленную на рис. 27.8 справа, в которой интерфейс пользователя, сервисы и доступ к данным перемежаются. Интерфейс пользователя и сервисы реализованы с помощью одних и тех же компонентов, нет четкого разделения между сервисами и базой данных системы. В таких условия определение компонентов, подлежащих распределению, практически невозможно.

В случаях, когда невозможно разделять наследуемую систему на распределяемые компоненты, следует применить альтернативный подход. Наследуемая система преобразуется в сервер, интерфейс пользователя реализуется на машине клиента, а промежуточное ПО обеспечивает взаимосвязь запросов, поступающих с машины клиента, с наследуемой системой. Этот подход показан на рис. 27.9.

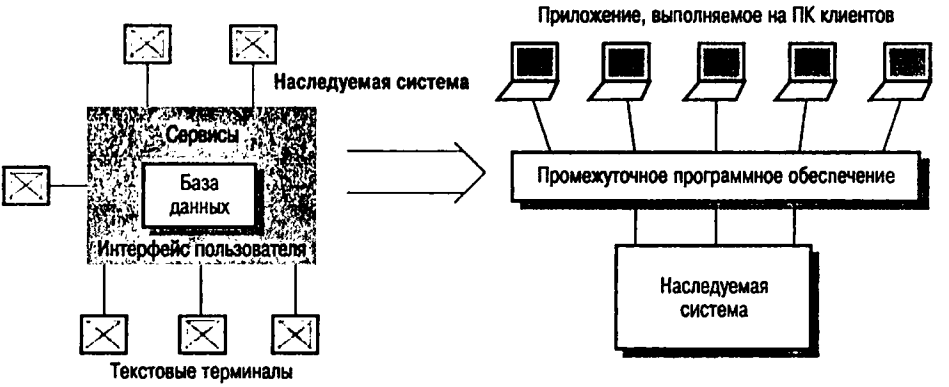


Рис. 27.9. Преобразование наследуемой системы в распределенную

Несмотря на интеграцию интерфейса пользователя и предоставляемых наследуемой системой сервисов, все-таки при планировании децентрализации лучше рассматривать их в качестве отдельных логических уровней (рис. 27.10).

1. Уровень представления отвечает за организацию вывода на экран информации для конечных пользователей системы.
2. Уровень проверки данных связан с управлением вводом-выводом данных, осуществляемым конечными пользователями.
3. Уровень управления взаимодействием определяется последовательностью операций конечных пользователей и порядком смены экранов, отображающихся на машинах конечных пользователей.
4. Уровень сервисов приложения отвечает за выполнение основных вычислений приложением.
5. Уровень базы данных отвечает за хранение и управление данными приложения.

Создавать распределенную базу данных для большинства наследуемых систем невыгодно, но для распределения других уровней существует целый ряд альтернатив, которые показаны на рис. 27.11. В простейшем случае компьютер клиента предоставляет только интерфейс пользователя, все другие уровни реализованы на сервере. Противоположный случай — сервер управляет только базой данных, все остальные операции выполняет машина клиента. Естественно, что эти варианты распределения не являются взаимоисключающими. Можно начать с распределения уровня представления, а при наличии ресурсов и времени можно распределить другие логические уровни. В главе 11 рассмотрены другие варианты распределенных систем.

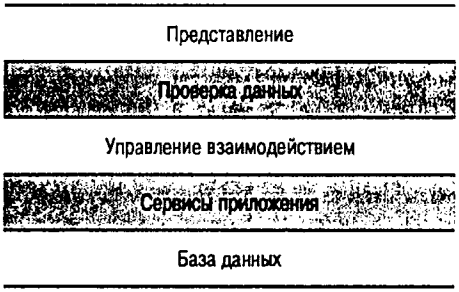


Рис. 27.10. Многоуровневая модель системы

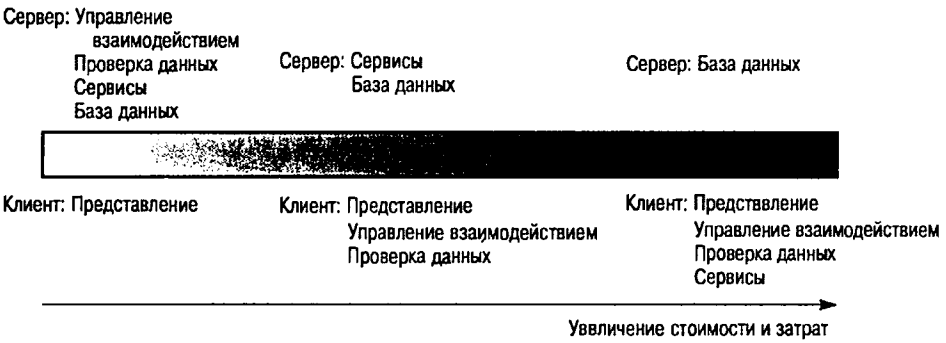


Рис. 27.11. Варианты распределенных систем

Когда наследуемая система представлена в виде сервера и доступ к ней осуществляется посредством промежуточного ПО (см. рис. 27.9), распределение системы можно начинать с варианта, показанного на рис. 27.11 слева, а затем постепенно переходить к варианту, показанному справа. При реализации новых сервисов они принимают на себя функции наследуемой системы на сервере, передавая управление операциями по обработке данных машине клиента. В конце концов такая постепенная переадресация функций приводит к тому, что большая часть изначальной наследуемой системы не используется, она принимает на себя только функции канала обслуживания базы данных для распределенной системы. При достижении этого этапа уже можно будет решать, оставить ли наследуемую систему либо заменить ее системой управления данными.

27.3.1. Распределение интерфейсов пользователя

Многие наследуемые системы введены в эксплуатацию еще до того, как были изобретены графические интерфейсы пользователей. Такие системы имеют интерфейсы, основанные на текстовых формах, которые выполняются на терминалах, способных выводить на экран только символьные изображения. Вычислительные средства таких терминалов относительно слабые, поэтому все основные вычислительные функции принимает на себя центральная ЭВМ.

При распределении интерфейсов пользователя применяются мощности локальных ПК, обеспечивающие графический интерфейс, который в большей мере отвечает потребностям пользователей. Функции интерфейса (представление данных, управление взаимодействием и проверка данных) переводятся на локальный ПК, а текстовый интер-

фейс заменяется графическим интерфейсом пользователя. Серверу остаются функции по обработке данных и реализации сервисов приложения.

Если наследуемая система обладает четкой структурой, в которой легко выделить интерфейс пользователя, ее можно преобразовать в систему с распределенными интерфейсами пользователя. Для этого следует перенести на компьютер клиента те компоненты системы, которые отвечают за взаимодействие с пользователем. Связь этих компонентов с основной программой осуществляется с помощью интерфейса, подобного изначальному текстовому.

Однако часто встречаются системы, в которых интерфейс и приложение интегрированы так, что невозможно вычленить код интерфейса. В этом случае можно прибегнуть к варианту распределения интерфейсов пользователя, который показан на рис. 27.12.

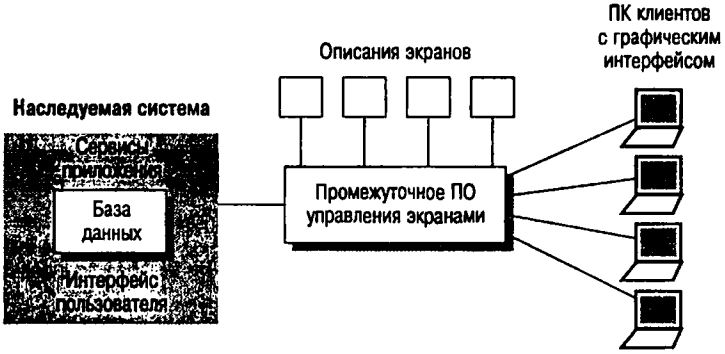


Рис. 27.12. Распределение пользовательских интерфейсов

Промежуточное ПО управления экранами (окнами), показанное на рис. 27.12, осуществляет связь с приложением и действует точно так же, как терминал пользователя. Это программное обеспечение использует описание каждого экрана для интерпретации и вывода данных на экран. В таком виде данные пересылаются на машину клиента, где они представляются с помощью графического интерфейса. В настоящее время процесс описания структуры интерфейса можно реализовать с помощью языка XML [326]. В этом случае не обязательно изменять наследуемую систему. Нужно только создать промежуточное ПО управления экранами и программу поддержки интерфейса на машине клиента.

Для реализации распределения пользовательских интерфейсов используются две стратегии.

1. Реализация интерфейса с помощью системы управления окнами, установленной на машине клиента и осуществляющей связь с сервером.
2. Реализация интерфейса пользователя с помощью Web-броузера.

В первом случае интерфейс создается с помощью подходящего языка программирования, например Java, или с помощью языка сценариев Visual Basic. Для реализации интерфейса на машине пользователя выполняются запросы к функциям операционной системе. Во втором случае для создания интерфейса на основе Web-страниц применяется язык HTML и Web-броузеры. Каждый подход имеет свои преимущества и недостатки, которые представлены в табл. 27.2.

Таблица 27.2. Преимущества и недостатки стратегий реализации распределенных пользовательских интерфейсов

Стратегия	Преимущества	Недостатки
Реализация с помощью системы управления окнами	Доступ ко всем функциям интерфейса пользователя. Улучшенная работа интерфейса пользователя	Зависимость от аппаратной платформы. Трудности согласования интерфейсов
Реализация с помощью Web-броузера	Независимость от аппаратной платформы. Снижение затрат на обучение работе с интерфейсом (интерфейс Web-броузера знаком всем). Легче добиться согласования интерфейсов	Более низкая производительность интерфейса. Возможности дизайна интерфейса ограничены возможностями Web-броузера

Переход с обычных интерфейсов на интерфейсы Web-броузеров приобрел такую популярность благодаря независимости от аппаратной платформы и широким возможностям Web-броузеров. Приложения на языке Java применимы для локальных вычислений на машине клиента, что позволяет сравнить эти интерфейсы с теми, которые основаны на системе управления окнами.

Большая трудность в преобразовании интерфейсов, основанных на текстовых формах, в графические состоит в том, что они различаются способами управления взаимодействием и проверки данных. В первом случае компьютерная система осуществляет управление взаимодействием и выполняет проверку данных сразу после поступления информации. Для этого необходимо, чтобы поля форм заполнялись в определенном порядке. В графическом интерфейсе пользователь произвольно выбирает интересующие его поля. Такое управление машина не может предсказать. Поэтому возникает дополнительный поток обмена данными между ПК пользователя и сервером. Проверка данных возможна лишь в случае полного заполнения всех полей, иначе работа системы замедляется за счет частых обменов данными между клиентом и сервером.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Стратегии модернизации программного обеспечения состоят из сопровождения ПО, эволюции системной архитектуры и реинжиниринга ПО.
- Существует ряд постоянных факторов (законы Лемана), влияющих на эволюцию программных систем. Эти законы появились благодаря практическим наблюдениям и дают необходимую основу для управления процессом сопровождения.
- Есть три вида сопровождения программных продуктов: с целью исправления ошибок, с целью адаптации ПО к рабочему окружению системы и с целью изменения или расширения функциональных возможностей системы.
- Затраты на сопровождение, как правило, превышают расходы на создание программной системы. Поскольку в большинстве компаний используются устаревшие системы, большая часть средств, выделяемых на ПО, уходит именно на сопровождение таких систем.
- Эволюция системной архитектуры подразумевает переход от централизованной структуры системы к распределенной.
- Общепринятой стратегией эволюции системной архитектуры является преобразование наследуемой системы в виде сервера и реализация распределенных интерфейсов пользователя, доступ к системным функциям осуществляется посредством специального промежуточного ПО.

Упражнения

27.1. Объясните, почему в "молодых" организациях-разработчиках постоянно возникают запросы на изменение программного обеспечения. Почему расходы на эти изменения неизбежно возрастают?

27.2. Объясните, на чем основаны законы Лемана. При каких условиях эти законы не будут действовать?

27.3. Опишите трудности, которые возникают при определении сложности сопровождения. Почему для оценивания будущего процесса сопровождения рекомендуется применять разные показатели сложности систем?

27.4. Допустим, вы менеджер по программному обеспечению компании, специализирующейся на разработке программ для нефтедобывающей промышленности, и вам дали задание определить факторы, влияющие на сопровождение систем, созданных компанией. Опишите, как вы организуете анализ процесса сопровождения с тем, чтобы найти систему показателей сопровождения, подходящую для вашей компании.

27.5. Объясните, почему преобразование в сервер наследуемой системы, реализованной на мэйнфрейме, является лишь краткосрочным решением проблемы эволюции системной архитектуры.

27.6. Опишите преимущества и недостатки распределения каждого системного слоя, показанного на рис. 27.10.

27.7. Два крупных международных банка с разными базами данных клиентов решили объединиться и предоставить доступ к объединенной базе данных из всех отделений банков. Предоставьте наиболее корректное решение для изменения ПО обоих банков, обоснуйте свой выбор и опишите процесс его реализации.

27.8. Будет ли, по вашему мнению, разработчик ПО, опираясь на свой профессиональный долг, создавать программный код, удобный для сопровождения, в том случае, если это не входит в список требований заказчика ПО?

28 Реинжениринг программного обеспечения

Цели

Цель настоящей главы — описание процесса реинжениринга, предполагающего повышение удобства эксплуатации программных систем. Прочитав главу, вы должны:

- знать, почему реинжениринг часто является наиболее выгодным решением для развития компьютерных систем;
- иметь представление о составляющих процесса реинжениринга;
- понимать различие между реинжинирингом программных систем и изменением системных данных, а также то, почему изменение данных является дорогостоящим процессом и требует много времени.

Содержание

В главах 26 и 27 дано описание наследуемых систем и различных стратегий развития программного обеспечения. Напомню, что наследуемой я называю старую систему, необходимую для поддержки текущей деловой активности организации, которая пока не может от нее отказаться. Организации во многом зависят от таких наследуемых систем, поэтому должны поддерживать их функционирование. В понятие эволюции наследуемой системы входят такие компоненты, как сопровождение, замена, архитектурная эволюция и реинжиниринг, изучением которого мы займемся в этой главе.

Реинжиниринг — это повторная реализация наследуемой системы в целях повышения удобства ее эксплуатации и сопровождения. В это понятие входят разные процессы, среди которых назовем повторное документирование системы, ее реорганизацию и реструктуризацию, перевод системы на один из более современных языков программирования, модификацию и модернизацию структуры и системных данных. При этом функциональность системы и ее архитектура остаются неизменными.

С технической точки зрения реинжиниринг — это решение "второго сорта" проблемы системной эволюции. Если учесть, что архитектура системы не изменяется, то сделать централизованную систему распределенной представляется делом довольно сложным. Обычно нельзя изменить язык программирования старых систем на объектно-ориентированные языки (например, Java или C++). Эти ограничения вводятся для сохранения архитектуры системы.

Однако с коммерческой точки зрения реинжиниринг часто принимается за единственный способ сохранения наследуемых систем в эксплуатации. Другие подходы к эволюции системы либо слишком дорогостоящие, либо рискованные. Чтобы понять причины такой позиции, следует рассмотреть проблемы, связанные с наследуемыми системами.

Код эксплуатируемых в настоящее время программных систем чрезвычайно огромен. В 1990 году Улрич (Ulrich, [336]) насчитал 120 млрд. строк исходного кода эксплуатируемых в то время программ. При этом большинство программ были написаны на языке COBOL, который лучше всего подходит для обработки данных в деловой сфере, и на языке FORTRAN. У этих языков достаточно ограниченные возможности в плане структуризации программ, а FORTRAN к тому же отличается ограниченной поддержкой структурирования данных.

Несмотря на постоянную замену подобных систем, многие из них все еще используются. С 1990 года отмечается резкое возрастание использования вычислительной техники в деловой сфере. При грубом подсчете можно говорить о 250 млрд. строк исходного кода, которые нуждаются в сопровождении. Большинство создано отнюдь не с помощью объектно-ориентированных языков программирования, многие из них функционируют все еще на мэйнфреймах.

Программных систем настолько много, что говорить о полной замене или радикальной реструктуризации их в большинстве организаций не приходится. Сопровождение старых систем действительно стоит дорого, однако реинжиниринг может продлить время их существования. Как отмечалось в главе 26, реинжиниринг систем выгоден в том случае, если система обладает определенной коммерческой ценностью, но дорога в сопровождении. С помощью реинжиниринга совершенствуется системная структура, создается новая документация и облегчается сопровождение системы.

По сравнению с более радикальными подходами к совершенствованию систем реинжиниринг имеет два преимущества.

1. *Снижение рисков.* При повторной разработке ПО существуют большие риски — высока вероятность ошибок в системной спецификации и возникновения проблем во время разработки системы. Реинжиниринг снижает эти риски.

2. *Снижение затрат.* Себестоимость реинжиниринга значительно ниже, чем разработка нового программного обеспечения. В статье [336] приводится пример системы, эксплуатируемой в коммерческой структуре, повторная разработка которой оценивалась в 50 млн. долларов. Для этой системы был успешно выполнен реинжиниринг стоимостью всего 12 млн. долларов. Приведенные цифры типичны: считается, что реинжиниринг в четыре раза дешевле, чем повторная разработка системы.

Реинжиниринг ПО тесно связан с реинжинирингом деловых процессов [153, 9*]. Последний означает преобразование бизнес-процессов для снижения количества излишних видов деятельности и повышения эффективности делового процесса. Обычно реинжиниринг бизнес-процессов предполагает внедрение новых программ для поддержки деловых процессов или модификацию существующих программ, при этом наследуемые системы существенно зависят от делового процесса. Такую зависимость следует выявлять как можно раньше и устранять, прежде чем начнется планирование каких-либо изменений в самом бизнес-процессе. Поэтому решение о реинжиниринге ПО может возникнуть, если наследуемую систему не удается адаптировать к новым деловым процессам путем изменений в обычном сопровождении системы.

Основное различие между реинжинирингом и новой разработкой системы связано со стартовой точкой начала работы над системой. При реинжиниринге вместо написания системной спецификации "с нуля" старая система служит основой для разработки спецификации новой системы. В статье [72] традиционная разработка ПО названа *разработкой вперед* (forward engineering), чтобы подчеркнуть различие между ней и реинжинирингом. Это различие проиллюстрировано на рис. 28.1. Традиционная разработка начинается с этапа создания системной спецификации, за которой следует проектирование и реализация новой системы. Реинжиниринг основывается на существующей системе, которая разработчиками изучается и преобразуется в новую.

Рис. 28.1. Традиционная разработка и реинжиниринг ПО

На рис. 28.2 показан возможный процесс реинжиниринга. В начале этого процесса имеем наследуемую систему, а в результате – структурированную и заново скомпонованную версию той же системы. Перечислим основные этапы этого процесса.

1. *Перевод исходного кода.* Конвертирование программы со старого языка программирования на современную версию того же языка либо на другой язык.
2. *Анализ программы.* Документирование структуры и функциональных возможностей программ на основе их анализа.
3. *Модификация структуры программы.* Анализируется и модифицируется управляющая структура программ с целью сделать их более простыми и понятными.

4. *Разбиение на модули.* Взаимосвязанные части программ группируются в модули; там, где возможно, устраняется избыточность. В некоторых случаях изменяется структура системы.
5. *Изменение системных данных.* Данные, с которыми работает программа, изменяются с тем, чтобы соответствовать нововведениям.

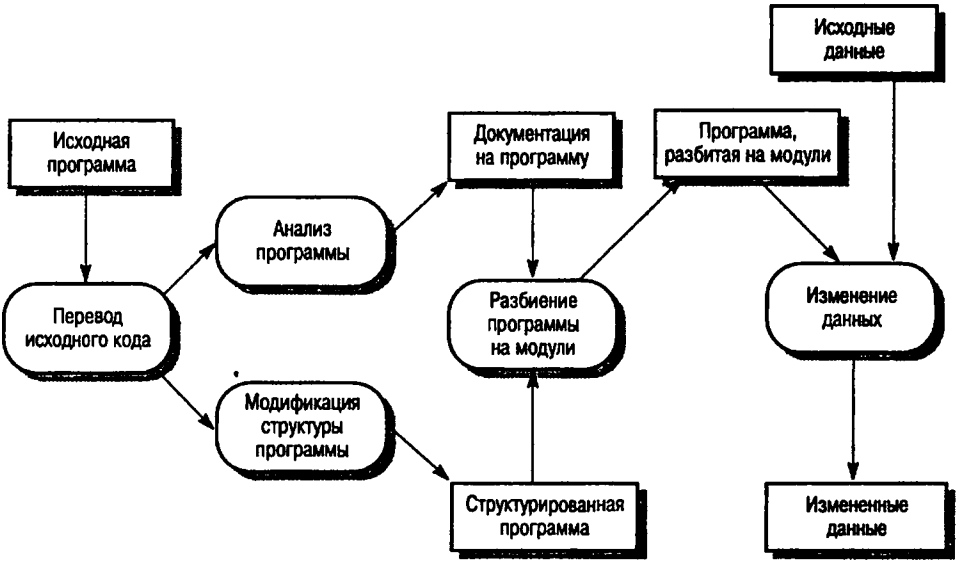


Рис. 28.2. Процесс реинжиниринга

При реинжиниринге программ необязательно проходить все стадии, показанные на рис. 28.2. Например, не всегда нужно переводить исходный код, если язык программирования, на котором написана программа, все еще поддерживается разработчиком компилятора. Если реинжиниринг проводится с помощью автоматизированных средств, то не обязательно восстанавливать документацию на программу. Изменение системных данных необходимо, если в результате реинжиниринга изменяется их структура. Однако реструктуризация данных в процессе реинжиниринга требуется всегда.

Стоимость реинжиниринга обычно определяется объемом выполненных работ. На рис. 28.3 показано несколько различных подходов к процессу реинжиниринга и динамика изменения стоимости работ для этих подходов.

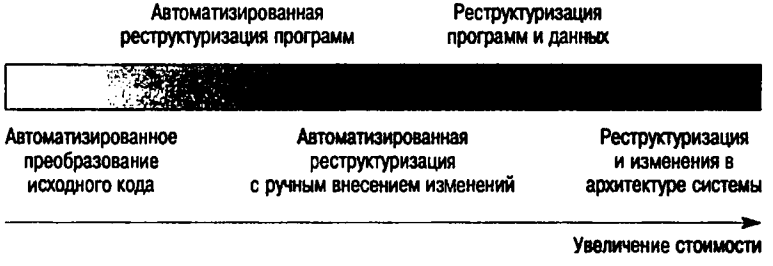


Рис. 28.3. Стоимость реинжиниринга

Кроме объема выполняемых работ, есть и другие факторы, обусловливающие стоимость реинжиниринга.

1. *Качество программного обеспечения, которое подвергается реинжинирингу.* Чем ниже качество программ и их документации (если она есть в наличии), тем выше стоимость реинжиниринга.
2. *Наличие средств поддержки процесса реинжиниринга.* Обычно реинжиниринг экономически выгоден, если применяются CASE-средства для автоматизированного внесения изменений в программы.
3. *Объем необходимого преобразования данных.* Стоимость процесса реинжиниринга возрастет при увеличении объема преобразуемых данных.
4. *Наличие необходимых специалистов.* Если персонал, который занимается сопровождением системы, не может выполнить реинжиниринг, это также может стать причиной повышения стоимости процесса. Вновь привлеченные специалисты потратят много времени на изучение системы.

Основным недостатком реинжиниринга принято считать то, что с его помощью систему можно улучшить только до определенной степени. Например, с помощью реинжиниринга невозможно функционально-ориентированную систему сделать объектно-ориентированной. Основные архитектурные изменения или полную реструктуризацию программ невозможно выполнить автоматически, что также увеличивает стоимость реинжиниринга. И, несмотря на то что реинжиниринг поможет улучшить сопровождение системы, все равно она будет намного хуже в сопровождении, чем новая, созданная с помощью современных методов инженерии ПО.

28.1. Преобразование исходного кода программ

Самый простой способ реинжиниринга программ – это автоматический перевод исходного кода с одного языка программирования на другой, более современный. При этом структура и организация программ остаются неизменными. Программа может переводиться как на обновленную версию исходного языка (например, с языка COBOL-74 на язык COBOL-85), так и на другой "не родственный" язык (например, с языка FORTRAN на C).

Причины перевода на другой язык могут быть следующие.

1. *Обновление платформы аппаратных средств.* В организации может быть принято решение по изменению аппаратной платформы. Новые аппаратные средства могут не поддерживать компиляторы исходного языка программ.
2. *Недостаток квалифицированного персонала.* Бывает, что для сопровождения программ на исходном языке невозможно найти достаточно квалифицированный персонал, особенно это касается программ, написанных на специфических языках, давно вышедших из употребления.
3. *Изменения политики организации.* Организация может принять решение о переходе на общий стандартный язык программирования, чтобы снизить затраты на сопровождение программных систем, поскольку сопровождение большого количества версий старых компиляторов невыгодно.

4. *Недостаточно средств поддержки старого ПО.* Поставщик компиляторов для старого языка программирования может уйти с рынка программных продуктов или прекратить поддержку своего продукта.

Процесс перевода исходного кода программ показан на рис. 28.4. Преобразование исходного кода будет эффективным только тогда, когда есть возможность выполнить основной перевод автоматически. Это может сделать либо специально созданная программа, либо коммерческая программа по конвертированию кода с одного языка в другой, либо система сопоставления с образцом. В последнем случае нужно создать список команд для описания перевода с одного языкового представления на другое. Параметризированные образцы исходного языка подвергаются сравнению и сопоставлению с такими же образцами в новом языке.

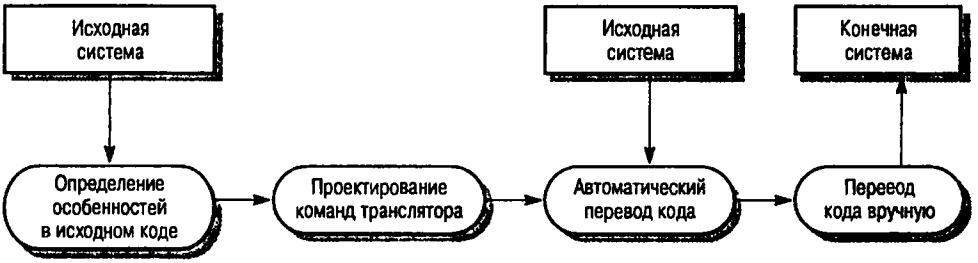


Рис. 28.4. Процесс преобразования программ

В некоторых случаях автоматизированный перевод становится невозможным. Структурные компоненты исходного кода могут не иметь соответствия в новом языке. Одна из причин этого в том, что исходный язык может содержать встроенные условные команды компиляции, которые не поддерживаются в новом языке. В такой ситуации придется настраивать и совершенствовать создаваемую систему вручную.

28.2. Анализ систем

Цель такого анализа – восстановление структуры и спецификации системы. Этот процесс не подразумевает изменения программ. Входными данными процесса анализа обычно служит исходный код системы. Однако зачастую даже он недоступен, тогда процесс анализа начинается с исполняемой программы.

Анализ систем не тождественен реинжинирингу систем. Целью анализа является определение архитектуры и спецификации системы на основе ее исходного кода. Целью реинжиниринга можно назвать создание усовершенствованной и удобной в сопровождении системы. Но, как показано на рис. 28.2, анализ системы может быть составной частью процесса реинжиниринга.

Схема процесса анализа системы приведена на рис. 28.5. Вначале с помощью автоматизированных средств проводится анализ структуры системы. В большинстве случаев этого недостаточно для воссоздания системной архитектуры. Требуется дополнительная работа с исходным кодом системы и с моделью ее структуры. Эта дополнительная информация сравнивается с данными, собранными во время автоматического анализа системы, и представляется в виде ориентированного графа, отображающего связи в исходном коде программ.

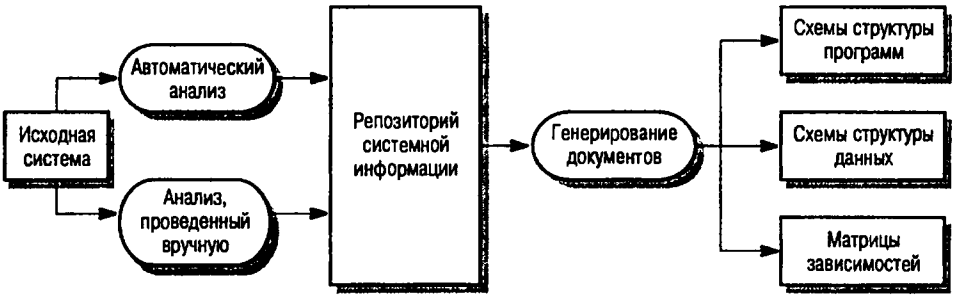


Рис. 28.5. Процесс анализа систем

Репозиторий системной информации служит для сравнения структуры графа и кода. На основе ориентированного графа можно получить такие документы, как схемы структуры программ и данных, а также матрицы зависимостей. Матрицы зависимостей показывают места определения в программах системных объектов и ссылки на них. Процесс разработки документации повторяющийся, так как информация о системной структуре используется для дальнейшего уточнения информации, которая хранится в системном репозитории.

В процессе анализа полезны разные средства просмотра программ, которые предоставляют различные системы представления программ и позволяют легко перемещаться по исходному коду. Например, с их помощью можно найти определения данных, а затем переместиться по коду к месту их использования. Описание некоторых подобных систем просмотра программ можно найти в [73, 263, 257].

После создания документации по системной архитектуре, в репозиторий вводится дополнительная информация, позволяющая восстановить системную спецификацию. Также обязательно ручное описание структуры системы. К сожалению, спецификацию невозможно создать автоматически из модели системы.

28.3. Совершенствование структуры программ

Если в процессе эксплуатации наследуемой системы возникла необходимость оптимизировать использование памяти и имеются проблемы с пониманием того, как она работает, это означает, что система плохо структурирована. Управляющая структура наследуемых систем обычно значительно усложнена множеством безусловных переходов и нечеткой логикой программного кода. Регулярное сопровождение системы также не способствует сохранению системной структуры. После частых изменений некоторые фрагменты кода становятся неиспользуемыми, однако это можно обнаружить только после тщательного анализа программы.

В листинге 28.1 показан пример того, как усложненная логика управления может сделать трудной для понимания достаточно простую программу. Программа написана на языке, подобном FORTRAN, который часто использовался для создания программ такого рода. Программа не стала понятнее даже после того, как я дал переменным осмысленные имена. Это программа управления обогревателем. Панельный переключатель имеет положения On (Включено), Off (Выключено) и Controlled (Регулирование). Если система находится в режиме регулирования, она включается или выключается в зависимости от термореле и установок таймера. Если обогреватель включен, переключатель Switch-heating выключает его, и наоборот.

Как правило, такая сложная логическая структура, как в коде листинга 28.1, образуется после внесения изменений в процессе сопровождения. Реализуя новые условия или связанные с ними действия, забывают об изменении структуры программы. Не задумываясь о перспективе, этот путь можно назвать кратчайшим и менее рискованным, так как он снижает вероятность возникновения большого количества ошибок в системе. Если же подумать о будущем, это решение приведет к трудному для понимания коду. Сложная структура кода может также появиться от желания программистов избежать дублирования кода. Ранее, когда на программы накладывалось требование ограничения памяти, это было обязательным условием.

Листинг 28.1. Программа с нечеткой логикой

```
Start: Get (Time-on, Time-off, Time, Setting, Temp, Switch)
       if Switch = off goto off
       if Switch = on goto on
       goto Cntrld
off: if Heating-status = on goto Sw-off
       goto loop
on: if Heating-status = off goto Sw-on
       goto loop
Cntrld: if Time = Time-on goto on
       if Time = Time-off goto off
       if Time < Time-on goto Start
       if Time > Time-off goto Start
       if Temp > Setting then goto off
       if Temp < Setting then goto on
Sw-off: Heating-status := off
       goto Switch
Sw-on: Heating-status := on
Switch: Switch-heating
loop: goto Start
```

В листинге 28.2 приведена та же самая программа, переписанная мной с использованием структурированных управляющих конструкций. Три положения переключателя (On, Off и Controlled) четко определены и связаны с соответствующим кодом. Язык Java не был использован при написании этой программы потому, что исходная программа не являлась объектно-ориентированной.

Листинг 28.2. Структурированная программа

```
loop
    -- Get получает значения переменных из окружения системы
    Get (Time-on, Time-off, Time, Setting, Temp, Switch) ;
    case Switch of
        when On => if Heating-status = off then
                Switch-heating ; Heating-status := on ;
            end if ;
        when Off=> if Heating-status = on then
                Switch-heating ; Heating-status :=off ;
            end if ;
        when Controlled =>
            if Time >= Time-on and Time <= Time-off then
                if Temp > Setting and Heating-status = on then
                    Switch-heating; Heating-status = off ;
                elsif Temp < Setting and Heating-status = off then
```

```
                    Switch-heating; Heating-status := on ;
                end if ;
            end if ;
    end case ;
end loop ;
```

В процессе реструктуризации программ можно также упрощать сложные условные операторы. В листинге 28.3 показан пример упрощения условного оператора, содержащего логический оператор отрицания not.

Листинг 28.3. Упрощение условия

```
-- Сложное условие
if not (A > B and (C < D or not (E > F) ) )...

-- Упрощенное условие
if A <= B and (C <= D or E > F)...
```

В известной работе [50] доказано, что любую программу можно переписать с помощью простых условных операторов if-then-else и циклов while-loop, при этом можно исключить все безусловные операторы перехода. Эта теорема является основой автоматической реструктуризации программ. Этапы такого преобразования программ показаны на рис. 28.6. Сначала программа представляется в виде ориентированного граф, после чего создается структурированная программа без использования операторов перехода.

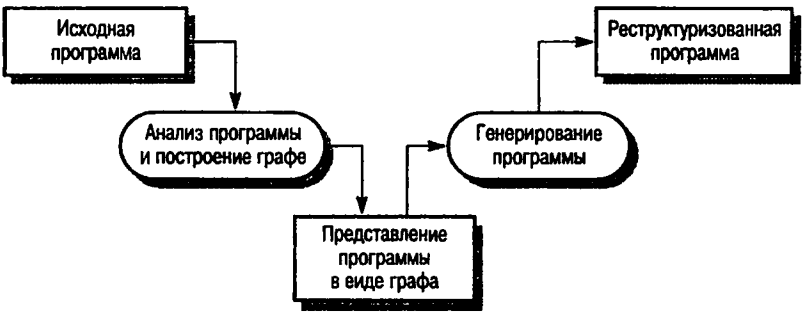


Рис. 28.6. Автоматическая реструктуризация программ

Созданный ориентированный граф показывает поток передачи управления в программе. К этому графу применяются методы упрощения и преобразования, в результате чего находятся и устраняются неиспользуемые части кода. После этого генерируется новая программа, при этом операторы безусловного перехода заменяются циклами и условными операторами. Такая программа может быть написана как на исходном языке, так и на любом другом (например, программу на языке FORTRAN можно конвертировать в программу на C).

Автоматизированный способ реструктуризации программ имеет свои проблемы.

1. *Потеря комментариев.* Если в программе есть встроенные комментарии, они будут утеряны в процессе реструктуризации.
2. *Утрата документации.* По той же причине обычно нарушается соответствие между новой программой и документацией на исходную программу. Однако в большинстве случаев это не так уж важно, поскольку документация и комментарии уже устарели.

3. *Жесткие требования к компьютерной технике.* Алгоритмы, встроенные в средства реструктуризации, отличаются высокой сложностью. Процесс реструктуризации больших программ, даже выполненный на современных быстродействующих компьютерах, будет занимать много времени.

Если программа находится под управлением данных и программные компоненты тесно связаны с используемыми структурами данных, реструктуризация кода не обязательно значительно улучшит программу. Если программа была написана с помощью редкого варианта языка программирования, стандартные средства преобразования структуры могут выполняться некорректно, поэтому неизбежно ручное вмешательство.

Иногда не стоит реструктуризировать все программы системы. Некоторые программы могут отличаться хорошим качеством, другие не подвергались большому количеству изменений, которые повредили бы их структуру. В работе [13] предлагается набор показателей для выявления тех программ, реструктуризация которых будет наиболее эффективной. Для этого можно использовать следующие показатели:

- интенсивность сбоев в работе программы;
- процентное соотношение кода, измененного на протяжении года;
- сложность компонентов.

При преобразовании структуры программ также следует учитывать степень соответствия программ или системных компонентов существующим стандартам.

28.4. Создание программных модулей

Это процесс реорганизации программы в целях объединения ее взаимосвязанных частей в отдельном модуле. После этого легче удалить избыточность в соответствующих компонентах, оптимизировать взаимосвязи и упростить интерфейс всей программы. Например, в программе по обработке сейсмографических данных все операции по графическому представлению данных можно собрать в один модуль. Если система будет распределенной, модули можно инкапсулировать как объекты, доступ к которым будет осуществляться через общий интерфейс.

В программной системе можно выделить различные типы модулей.

1. *Абстракции данных.* Это абстрактные типы данных, которые создаются путем объединения данных с компонентами их обработки. Этот тип модулей рассмотрен в разделе 28.4.1.
2. *Аппаратные модули.* Тесно связаны с абстракцией данных и объединяют все функции, управляющие отдельными аппаратными устройствами.
3. *Функциональные модули.* Объединяют все функции, которые выполняют сходные или взаимосвязанные задачи. Например, в один модуль можно объединить все функции, выполняющие ввод данных и их проверку. Этот подход применяется там, где создание абстракций данных невыгодно.
4. *Модули поддержки отдельных процессов.* В них сгруппированы все функции и данные, отвечающие за поддержку отдельного бизнес-процесса. Например, в библиотечной системе присутствует модуль, объединяющий все функции, отвечающие за выдачу и возврат книг.

Разбиение программы на модули обычно выполняется вручную путем проверки и правки кода. Для этого следует, прежде всего, определить взаимосвязи между компонентами и изучить способ их взаимодействия. Полностью автоматизировать этот процесс нельзя, даже если привлечь средства просмотра и визуализации программ.

28.4.1. Создание абстракций данных

Чтобы сберечь дисковую память, многие из наследуемых систем работают на основе совместно используемых массивов и общих областей данных. Это значит, что информация в этих областях полностью доступна и различные части системы используют ее по-своему. Изменение общих областей данных экономически невыгодно из-за высокой стоимости анализа влияния этих изменений на использование данных.

Именно для снижения стоимости таких изменений можно использовать разбиение программы на модули, построенные на основе абстракций данных. Абстракции данных (т.е. абстрактные типы данных) группируют сами данные и способы их обработки, что делает их более изменяемыми. Абстракции данных скрывают способ представления данных и обеспечивают доступ к ним. При хорошо разработанном интерфейсе модуля данных такие изменения типов данных не повлияют на другие части программы.

Чтобы преобразовать общие используемые области данных в объекты или абстрактные типы данных, следует выполнить ряд действий.

1. Провести анализ общих областей данных для выявления логических структур данных. Случается, что одну область используют данные нескольких разных типов. Такие ситуации следует выявлять и реконструировать.
2. Создать абстрактный тип данных или объект для каждой абстракции. Если в языке программирования нет способов сокрытия данных, можно имитировать абстрактный тип данных путем написания соответствующих функций, обеспечивающих обновление и доступ ко всем полям записей данных.
3. Осуществить поиск всех ссылок на данные с использованием системы просмотра программ или генератора перекрестных ссылок. Заменить эти ссылки соответствующими функциями.

На первый взгляд эти действия покажутся достаточно простыми, хотя и отнимающими много времени. Однако в действительности все гораздо сложнее из-за разных способов использования области совместных данных. В более старых версиях языков типа FORTRAN, у которых довольно ограниченный набор функций по структурированию данных, программисты могли разработать достаточно сложные стратегии управления данными с помощью совместно используемых массивов. Последующие проблемы вытекают из косвенной адресации совместно используемых структур, а также из адресации со смещением.

Проблемы другого рода возникают, если вычислительная машина, на которой выполнялась исходная программа, имеет ограниченную память. В этом случае программисты в разных частях программы могли использовать одну область данных для хранения разных типов данных. Такие явления распознаются только после детального статического и динамического анализа программ.

28.5. Изменение данных

До сих пор все обсуждаемые изменения касались в основном программ и систем. Однако в некоторых случаях придется столкнуться с проблемой изменения данных. Хранение, структура и формат данных, с которыми работает наследуемая система, должны изме-

ниться, чтобы соответствовать изменениям в программном обеспечении. Изменение данных — это процесс анализа и реорганизации структуры данных, а иногда еще и изменение значений системных данных.

В принципе, если функциональность системы осталась прежней, изменения данных не требуется. Однако существует ряд причин, которые вынуждают изменять данные (так же, как и программы) наследуемой системы.

1. *Нарушение данных.* С течением времени качество данных снижается. Изменения данных становятся причиной новых ошибок, возможно дублирование значений, изменения во внешнем окружении системы могут не найти адекватного отражения в данных. Эти явления неизбежны, так как время существования данных бывает достаточно большим. Например, персональные данные в банковской системе появляются с созданием нового счета и существуют, по меньшей мере, в течение всей жизни клиента. При изменении обстоятельств у клиента банковские данные должны обновляться, что не всегда происходит корректно. Реинжиниринг системы уменьшает эти трудности, что лишний раз подтверждает его необходимость.
2. *Программные ограничения.* При разработке систем многие программисты включают в программы ограничения на количество обрабатываемых данных. Но согласно современным требованиям программы должны обрабатывать значительно больше данных, чем было предусмотрено изначально. Именно для устранения подобных ограничений может понадобиться изменение данных. В книге [296] приведен пример системы управления ценными бумагами, которая была способна обрабатывать до 99 транзакций за одну операцию. В компании, где эта система использовалась, осуществлялось управление 2000 транзакций, что вызвало необходимость в создании 23 копий системы. По этой причине впоследствии компания приняла решение о реинжиниринге системы и изменении данных.
3. *Эволюция системной архитектуры.* При переходе с централизованной системы на распределенную ядром архитектуры должна стать система управления данными с удаленным доступом. Для перемещения данных из отдельных файлов на сервер системы управления базой данных (СУБД) может потребоваться большая работа по изменению этих данных.

Как и в случае с реинжинирингом программ, изменение данных имеет свои подходы и методы, которые перечислены в табл. 28.1.

Таблица 28.1. Методы изменения данных

Метод	Описание
Чистка данных	Устраняется дублирование, стирается избыточная информация, ко всем записям применяется единый формат. Все это, как правило, не влечет за собой никаких изменений в программах
Расширение возможностей обработки данных	Данные и связанные с ними программы подвергаются реинжинирингу для устранения ограничений на обработку данных. Например, увеличивается длина полей, увеличиваются верхние границы массивов и т.п. Также вносятся соответствующие изменения в программы. После этого данные обычно перезаписываются и очищаются
Миграция данных	Данные переводятся под управление современной СУБД. Этот подход проиллюстрирован на рис. 28.7

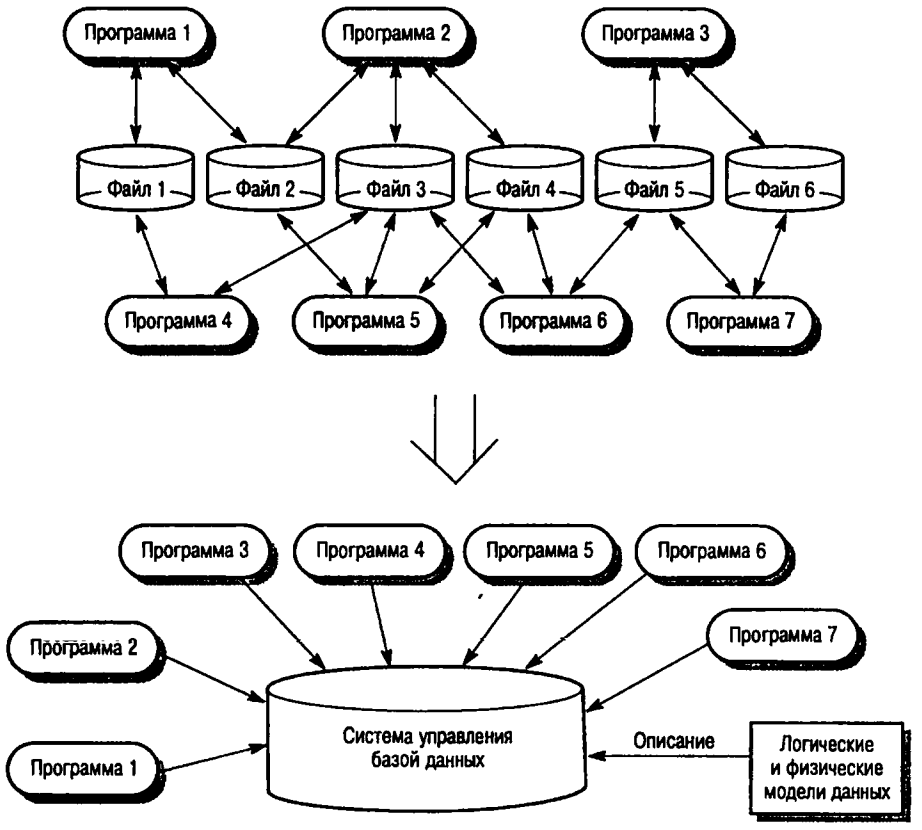


Рис. 28.7. Миграция данных

В статье [293] описаны некоторые проблемы с данными, возникающие в наследуемых системах, состоящих из нескольких программ коллективного пользования.

1. *Проблема именования данных.* Имена могут быть зашифрованы и трудны для восприятия. Одному и тому же логическому элементу в разных программах могут присваиваться разные имена. С другой стороны, одно и то же имя в разных программах используется для именования различных элементов.
2. *Проблема длины полей.* Возникает в тех случаях, когда длина поля определена непосредственно в программе. Одному и тому же элементу записи может быть определена разная длина в разных частях программы, либо длина поля слишком мала для представления текущих данных.
3. *Проблема организации записей.* Записи, относящиеся к одному и тому же элементу, в разных программах могут быть представлены по-разному. Обычно эта проблема возникает с такими языками программирования, как COBOL, где физическая организация записей определяется программистом. В языках типа C++ или Java такой проблемы не существует, так как физической организацией записей занимается компилятор.

4. *Проблема констант.* Константы (литеральные величины), например налоговые ставки, часто определены в программе, что затрудняет создание символьных ссылок на них.
5. *Отсутствие словаря данных.* Часто отсутствует словарь данных, в котором отображены применяемые имена, их представления и использование.

Если определения данных несовместимы или противоречивы, их значения могут храниться и обрабатываться некорректно. Примеры несовместимости и противоречивости данных приведены в табл. 28.2, взятой из [293]. После изменения определений данных их значения преобразуются так, чтобы соответствовать новой структуре данных.

Таблица 28.2. Примеры несовместимости и противоречивости значений данных

Данные	Описание
Различные значения по умолчанию	В различных программах одному и тому же логическому элементу данных могут быть присвоены разные значения по умолчанию. Это вызывает трудности в работе программ, которые обрабатывают эти данных. Особая проблема: недопустимое значение присваивается по умолчанию как допустимое
Различные единицы измерения	Разные программы представляют одинаковую информацию в разных единицах измерения. Например, в США и Великобритании вес может измеряться в фунтах (если взять более старую программу) и в килограммах (в современных системах). Проблема такого же рода возникла в Европе с введением единой валюты. Пришлось изменять системы, рассчитанные на работу с национальной валютой, для того чтобы они смогли работать с евро
Несовместимость правил проверки данных	В разных программах различные правила проверки данных. Данные, приемлемые для одной программы, могут не восприниматься другой. Особая проблема возникает с архивными данными, которые не обновлялись в соответствии с изменениями правил проверки
Противоречия в семантике представлений	Программы могут присваивать значения в зависимости от способа представления элементов. Например, в некоторых программах текст в верхнем регистре означает адрес. Программы используют различные соглашения о представлении данных и поэтому могут не воспринимать данные, хотя они и будут верными
Несогласованность при обработке отрицательных величин	В некоторых программах величинам, которые должны быть всегда положительными, не может быть присвоено отрицательное значение. Другие программы при предъявлении отрицательных величин могут конвертировать их в положительные и т.д.

Перед изменением данных необходимо провести подробный анализ программ, которые работают с этими данными. Главная цель анализа — определение в программе деклараций функций, выявление литеральных величин, требующих заменены на именованные константы, поиск встроенных правил проверки данных. При анализе помогают такие средства, как анализаторы перекрестных ссылок и сопоставление с образцом. Для фиксации мест ссылок на элементы данных и изменений, которые там требуются, удобно создать набор таблиц регистрации изменений, которые содержат описание всех этапов изменения данных. Схема процесса изменения данных показана на рис. 28.8.

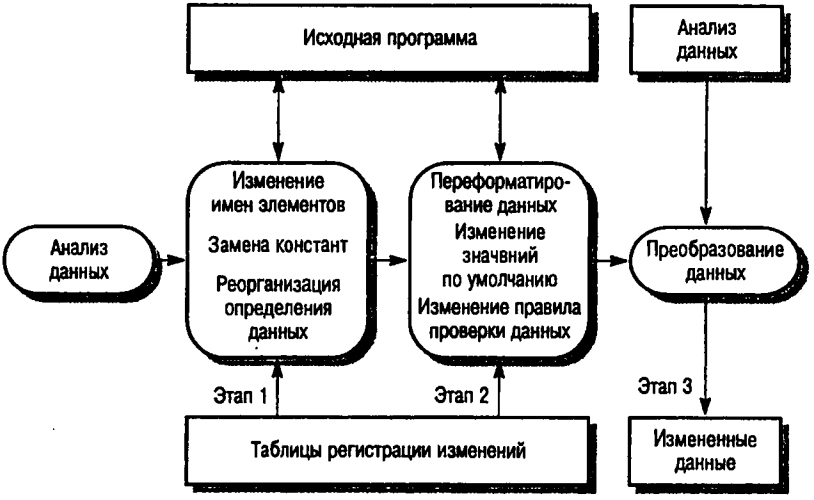


Рис. 28.8. Процесс изменения данных

На первом этапе процесса изменения данных модифицируются определения данных. На сами данные такая модификация не оказывает влияния. Чтобы автоматизировать этот процесс, можно использовать системы сопоставления с образцом, например awk [5], которые помогают находить и заменять определения, или же можно создать XML-определения данных [326] и использовать их для управления средствами конвертирования данных. Несмотря на это, ручной работы над данными практически невозможно избежать. Если ставится цель улучшить понимаемость определений данных, то работу можно остановить на этой стадии. Если же имеются проблемы со значениями данных, описанные выше, следует начать второй этап процесса изменения данных.

После второго этапа обязательно идет третий – преобразование данных. Обычно это очень дорогостоящий процесс. Для его реализации создаются программы, аккумулирующие информацию о старой и новой структурах данных. Здесь опять применяется система сопоставления с образцом.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Целью реинжиниринга систем является улучшение системных структур. В результате снижаются расходы на сопровождение систем.
- В процесс реинжиниринга систем входят такие этапы, как перевод исходного кода, анализ систем, улучшение структуры программ, разбиение программ на модули и изменение данных.
- Перевод исходного кода — это автоматизированный перевод программы, созданной с помощью одного языка программирования, на другой. Это необходимо, если исходный язык программирования устарел.
- В процессе анализа системы на основе исходного кода восстанавливаются системные архитектура и спецификация. Для поддержки процесса анализа используются средства просмотра программ.
- Для улучшения структуры программ такие неструктурные управляющие элементы, как операторы безусловного перехода, заменяются условными операторами и циклами.

- Разбиение на модули реорганизует исходные программы в совокупность модулей, объединяющих группу взаимосвязанных элементов. Благодаря этому программы легче понимать и изменять.
- Изменение данных применяется в том случае, когда в наследуемой системе возникают противоречия в управлении данными. Целью изменения данных может быть преобразование всех программ для работы с единой базой данных.
- Если данные нужно преобразовать в новый формат, стоимость замены данных резко возрастает.

Упражнения

28.1. В каких случаях необходимо заменить старое ПО новым вместо его реинжиниринга?

28.2. Сравните управляющие структуры (циклы и условные операторы) в двух любых известных вам языках программирования. Кратко опишите процесс перевода управляющих структур с одного языка на другой.

28.3. Переведите процедуру, приведенную в листинге 28.4, в эквивалентную структурированную, выполнив все необходимые для этого действия.

28.4. Напишите ряд указаний по определению модулей в неструктурированной программе.

28.5. Предложите значащие имена для переменных, которые используются в листинге 28.4, и создайте словарь данных для этих имен.

28.6. Какие трудности возникнут при переносе данных из СУБД одного типа в СУБД другого типа (например, из иерархической базы данных в реляционную или из реляционной в объектно-ориентированную)?

28.7. Объясните, почему невозможно восстановить системную спецификацию путем автоматического анализа исходного кода системы.

28.8. Приведите примеры для описания проблем, связанных с нарушением данных при их чистке.

28.9. Проблема 2000 года (когда даты представлены с помощью двух цифр) стала одной из основных причин для многих организаций внести коррективы в сопровождение программ. Как это повлияло на процесс изменения данных?

Листинг 28.4. Неструктурированная программа

```
routine BS (K, T, S, L)
B:= 1
NXT: if S >= B goto CON
L = -1
goto STP
CON: L := INTEGER(B / S)
L := INTEGER((B + S) / 2)
If T (L) = K then return
If T (L) > K then goto GRT
B := L + 1
goto NXT
GRT: S := L-1
goto NXT
STP: end
```

29

Управление конфигурациями

Цели

Цель настоящей главы – описание процесса управле ния программным кодом и документацией модифи цируемых программных систем. Прочитав главу, вы должны:

- понимать значение управления конфигурацией ПО;
- знать о четырех основных процессах управления конфигурацией: планирование управления, управ ление изменениями, управление версиями и сбор кой системы;
- иметь представление о применении CASE средств для поддержки процесса управления конфигурацией.

Содержание

Управление конфигурацией — это процесс разработки и применения стандартов и правил по управлению эволюцией программных продуктов. Эволюционирующие системы нуждаются в управлении по той простой причине, что в процессе их эволюции создается несколько версий одних и тех же программ. В эти версии обязательно вносятся некоторые изменения, исправляются ошибки предыдущих версий; кроме того, версии могут адаптироваться к новым аппаратным средствам и операционным системам. При этом в разработке и эксплуатации могут одновременно находиться сразу несколько версий. Поэтому нужно четко отслеживать все вносимые в систему изменения.

Процедуры управления конфигурацией регулируют процессы регистрации и внесения изменений в систему с указанием измененных компонентов, а также способы идентификации различных версий системы. Средства управления конфигурацией применяют для хранения всех версий системных компонентов, для компоновки из этих компонентов системы и для отслеживания поставки заказчикам разных версий системы.

Как указывалось в главе 24, управление конфигурацией нередко рассматривается как часть общего процесса управления качеством. Поэтому иногда одно и то же лицо может отвечать как за управление качеством, так и за управление конфигурацией. Но обычно разрабатываемая программная система сначала контролируется командой по управлению качеством, которая проверяет ПО на соответствие определенным стандартам качества. Далее ПО передается команде по управлению конфигурацией, которая контролирует изменения, вносимые в систему.

Существует много причин, объясняющих наличие разных конфигураций одной и той же системы. Различные версии создаются для разных компьютеров или операционных систем, включающих специальные функции, нужные заказчикам, и т.д. (рис. 29.1). Менеджеры по управлению конфигурацией обязаны следить за различиями между разными версиями, чтобы обеспечить возможность выпуска следующих вариантов системы и своевременную поставку нужных версий соответствующим заказчикам.

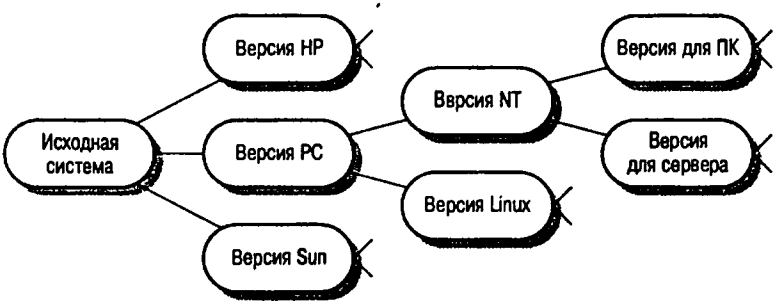


Рис. 29.1. Семейство версий системы

Процесс управления конфигурацией и связанная с ним документация должны подчиняться определенным стандартам. В качестве примера можно привести стандарт IEEE 828-1983, определяющий составление планов управления конфигурацией. Каждая организация должна иметь справочник, в котором указаны эти стандарты, либо они должны входить в общий справочник стандартов качества. Общенациональные или международные стандарты могут быть также использованы как основа для разработки детализированных специальных норм и стандартов для конкретных организаций. За основу можно взять любой тип стандарта, поскольку все они содержат описания однотипных процессов. Для сертификации качества своих программных продуктов организация должна придержи-

ваться официальных стандартов управления конфигурацией, которые приведены в стандартах ISO 9000 [178] и в модели оценки уровня развития SEI [273].

При традиционной разработке ПО в соответствии с каскадной моделью (см. главу 3) разрабатываемая система попадает в группу по управлению конфигурацией уже после полного завершения разработки и тестирования ПО. Именно такой подход лежит в основе стандартов управления конфигурацией, которые, в свою очередь, обусловливают необходимость использования для разработки систем моделей, подобных каскадной [38]. Поэтому упомянутые стандарты не в полной мере подходят при использовании таких методов разработки ПО, как эволюционное прототипирование и пошаговая разработка. В этой ситуации некоторые организации изменили подход к управлению конфигурацией, сделав возможным параллельную разработку и тестирование системы. Такой подход основан на регулярной (иногда ежедневной) сборке системы из ее компонентов.

1. Устанавливается время, к которому должна быть завершена поставка компонентов системы (например, к 14.00). Программисты, работающие над новыми версиями компонентов, должны предоставить их к указанному времени. Работу над компонентами не обязательно завершать, достаточно представить основные рабочие функции для проведения тестирования.
2. Создается новая версия системы с новыми компонентами, которые компилируются и связываются в единую систему.
3. После этого система попадает к группе тестирования. В то же время разработчики продолжают работу над компонентами, добавляя новые функции и исправляя ошибки, обнаруженные в ходе предыдущего тестирования.
4. Дефекты, замеченные при тестировании, регистрируются, соответствующий документ пересылается разработчикам. В следующей версии компонента эти дефекты будут учтены и исправлены.

Основным преимуществом ежедневной сборки системы является возможность выявления ошибок во взаимодействиях между компонентами, которые в противном случае могут накапливаться. Более того, ежедневная сборка системы поощряет тщательную проверку компонентов. Разработчики работают под давлением: нельзя прерывать сборку систем и поставлять неисправные версии компонентов. Поэтому программисты неохотно поставляют новые версии компонентов, если они не были предварительно тщательно проверены. Таким образом, на тестирование и исправление ошибок ПО уходит меньше времени.

Для ежедневных сборок системы требуется достаточно строгое управление процессом изменений, позволяющее отслеживать проблемы, которые выявляются и исправляются в ходе тестирования. Кроме того, в результате возникает множество версий компонентов системы, для управления которыми необходимы средства управления конфигурацией.

29.1. Планирование управления конфигурацией

В плане управления конфигурацией представлены стандарты, процедуры и мероприятия, необходимые для управления. Отправной точкой создания такого плана является набор общих стандартов по управлению конфигурацией, применяемых в организации-разработчике ПО, которые адаптируются к каждому отдельному проекту. Обычно план управления конфигурацией имеет несколько разделов.

1. Определение контролируемых объектов, подпадающих под управление конфигурацией, а также формальная схема определения этих объектов.
2. Перечень лиц, ответственных за управление конфигурацией и за поставку контролируемых объектов в команду по управлению конфигурацией.
3. Политика ведения управления конфигурацией, т.е. процедуры управления изменениями и версиями.
4. Описание форм записей о самом процессе управления конфигурацией.
5. Описание средств поддержки процесса управления конфигурацией и способов их использования.
6. Определение базы данных конфигураций, применяемой для хранения всей информации о конфигурациях системы.

Распределение обязанностей по конкретным исполнителям является важной частью плана. Необходимо четко определить ответственных за поставку каждого документа или компонента ПО для команд по управлению качеством и конфигурацией. Лицо, отвечающее за поставку какого-либо документа или компонента, должно отвечать и за их разработку. Для упрощения процедур согласования удобно назначать менеджеров проекта или ведущих специалистов команды разработчиков ответственными за все документы, созданные под их руководством.

29.1.1. Определение конфигурационных объектов

В процессе разработки больших систем создаются тысячи различных документов. Большинство из них — это текущие рабочие документы, связанные с различными этапами разработки ПО. Есть также внутренние записки, протоколы заседаиия рабочих групп, проекты планов и предложений и т.п. Такие документы представляют разве что исторический интерес и не нужны для дальнейшего сопровождения системы.

Для планирования процесса управления конфигурацией необходимо точно определить, какие проектные элементы (или классы элементов) будут объектами управления. Такие элементы называются *конфигурационными элементами*. Как правило, они представляют собой официальные документы. Конфигурационными элементами обычно являются планы проектов, спецификации, схемы системной архитектуры, программы и наборы тестовых данных. Кроме того, управлению подлежат все документы, необходимые для будущего сопровождения системы.

В процессе управления конфигурацией каждому документу необходимо присвоить уникальное имя, причем отображающее связи с другими документами. Для этого используется иерархическая система имен, где они имеют, например, такой вид:

PLC-TOOLS/ПРАВКА/ФОРМЫ/ОТОБРАЖЕНИЕ/ИНТЕРФЕЙСЫ/КОД
PLC-TOOLS/ПРАВКА/СПРАВКА/ЗАПРОС/ОКНО_СПРАВКИ/FR-1

Начальная часть имени — это название проекта PLC-TOOLS. В проекте разрабатываются четыре отдельных средства (рис. 29.2). Имя средства используется в следующей части имени. Каждое средство создается из именованных модулей. Такое разбиение продолжается до тех пор, пока не появится ссылка на официальный документ базового уровня. Листья дерева иерархии документов являются официальными документами проекта. На рис. 29.2 показано, что для каждого объекта требуется три формальных документа. Это описание объектов (документ ОБЪЕКТЫ), код компонента (документ КОД) и набор тестов для этого кода (документ ТЕСТЫ).

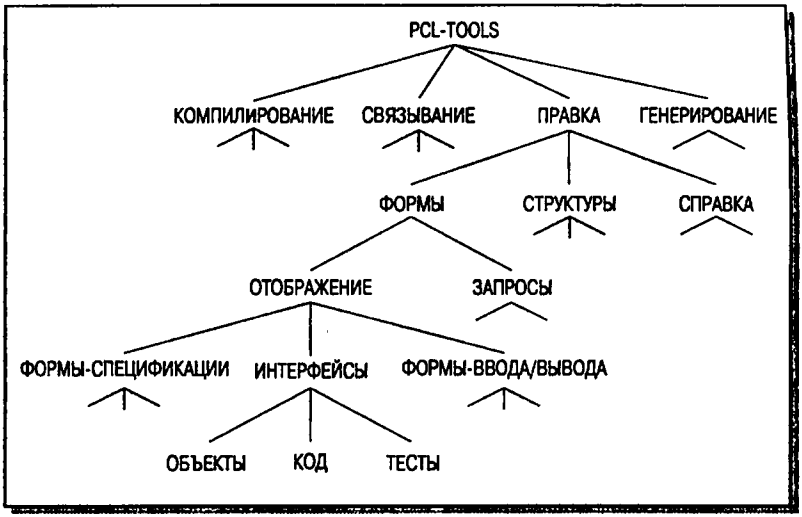


Рис. 29.2. Иерархия конфигурации

Подобные схемы имен основаны на структуре проекта, когда имена соотносятся с соответствующими проектными компонентами. Такой подход к именованию документов порождает определенные проблемы. Например, снижается возможность повторного использования компонентов. Обычно в таких случаях из схемы берутся копии компонентов, которые можно повторно использовать, и переименовываются в соответствии с новой областью применения. Другие проблемы могут появиться, если эта схема именования документов используется как основа структуры хранения компонентов. Тогда пользователь должен знать названия документов, чтобы найти нужные компоненты, при этом не все документы одного типа (например, по проектированию) хранятся в одном месте. Также могут встретиться трудности при установлении соответствия между схемой имен и схемой идентификации, используемой в системе управления версиями.

29.1.2. База данных конфигураций

Такая база данных используется для хранения всей информации о системных конфигурациях. Основными функциями базы данных конфигураций являются поддержка оценивания влияния планируемых изменений в системе и предоставление информации о процессе управления конфигурацией. Задание структуры базы данных конфигураций, определение процедур записи и поиска информации в этой базе данных — все это является частью процесса планирования управления конфигурацией.

Информация, заключенная в базе данных конфигураций, должна помочь ответить на ряд вопросов, среди которых основными и часто запрашиваемыми будут следующие.

- Каким заказчикам поставлена определенная версия системы?
- Какие аппаратные средства и какая операционная система необходимы для работы данной версии системы?
- Сколько было выпущено версий данной системы и когда?
- На какие версии системы повлияют изменения, вносимые в определенный компонент?
- Сколько запросов на изменения было реализовано в данной версии?
- Какое количество ошибок было зарегистрировано в данной версии системы?

В идеале база данных конфигураций должна быть объединена с системой управления версиями, которая создается для хранения и управления формальными проектными документами. Такой подход, к тому же поддерживаемый некоторыми интегрированными CASE-средствами, предоставляет возможность связать изменения, вносимые в систему, и с документами, и с теми компонентами, которые подверглись изменениям. В этом случае упрощается поиск измененных компонентов, поскольку установлены связи между документами (например, между документами по системной архитектуре и кодом программ) и этими связями можно управлять.

Однако многие организации вместо использования интегрированных CASE-средств для управления конфигурацией рассматривают базу данных конфигураций как отдельную систему. Конфигурационные элементы могут храниться в отдельных файлах или в системе управления версиями, например RCS (известная система управления версиями для Unix [335]). В этом случае в базе данных конфигураций хранится информация о конфигурационных элементах и ссылки на имена соответствующих файлов в системе управления версиями. Несмотря на относительную дешевизну и гибкость такого подхода, основным недостатком его является то, что конфигурационные элементы могут быть изменены без внесения необходимых записей в базу данных. Поэтому нельзя гарантировать, что в базе данных конфигураций содержится обновленная и корректная информация о состоянии системы.

29.2. Управление изменениями

Изменения в больших программных системах неизбежны. Как отмечалось в предыдущих главах, в течение жизненного цикла системы изменяются пользовательские и системные требования, а также приоритеты и запросы организаций. Процесс управления изменениями и соответствующие CASE-средства предназначены для того, чтобы зарегистрировать изменения и внести их в систему наиболее эффективным способом.

Процесс управления изменениями (листинг 29.1) начинается после того, как программное обеспечение или соответствующая документация передается команде по управлению конфигурацией. Он может начаться во время тестирования системы или даже после ее поставки заказчику. Процедуры управления изменениями создаются для обеспечения корректного анализа необходимости изменений и их стоимости, а также для контроля за вносимыми изменения.

Листинг 29.1. Процесс управления изменениями

```
Запрос на изменение, заполнение формы запроса
Анализ запроса
if изменение допустимо then
    Оценка способа внесения изменения
    Оценка стоимости изменения
    Запись запроса в базу данных
    Передача запроса группе контроля за изменениями
    if запрос принят then
        repeat
            внесение изменений в ПО
            регистрация изменений
            передача измененного ПО группе управления качеством
        until качество ПО соответствует нормам
        создание ноной версии системы
    else
        запрос на изменение отвергнут
else
    запрос на изменение отвергнут
```

Первым этапом в процессе управления изменениями является заполнение формы запроса на изменения, в которой указываются те изменения, которые планируется внести в систему. В форме запроса также приводятся рекомендации относительно изменений, предварительная оценка затрат и даты запроса, его утверждения, внедрения и проверки. Также форма может включать раздел, в котором указывается способ выполнения изменения. Запросы на изменения регистрируются в базе данных конфигураций. Таким образом, команда управления конфигурациями может следить за выполнением изменений, а также контролировать изменения определенных программных компонентов.

Во врезке 29.1 приведен пример заполненной формы запроса на изменения. Такая форма обычно определяется на этапе планирования управления конфигурацией. По условиям некоторых контрактов (например, в контрактах с правительственными органами), такая форма должна соответствовать стандартам заказчика.

Врезка 29.1. Форма запроса на изменения

Проект. Proteus/PCL-Tools **Номер.** 23/94
Лицо, заполняющее запрос. Соммервиль **Дата.** 1/12/98
Требуемые изменения. После выбора компонента структуры необходимо отображение имени файла, в котором он хранится.
Лицо, осуществляющее анализ запроса. Г. Дин
Дата анализа. 10/12/98
Изменяемые компоненты. Display-Icon.Select, Display-Icon.Display
Связанные компоненты. FileTable
Оценка изменения. Изменение достаточно легко выполнить из-за наличия таблицы имен файлов. Требуется вставить соответствующее поле. Изменений связанных компонентов не требуется.
Уровень приоритета. Низкий
Выполнение изменения.
Оценка затрат. 0,5 дня
Дата передачи в группу контроля за изменениями. 15/12/98
Дата принятия решения группой контроля за изменениями. 1/2/99
Решение группы контроля за изменениями. Принять изменение. Будет реализовано в версии 2.1.
Кто вносит изменения.
Дата внесения изменений.
Дата передачи запроса в группу управления качеством.
Решение группы управления качеством.
Дата передачи запроса в группу по управлению конфигурацией.
Примечание.

Сразу после представления заполненной формы запроса проводится проверка необходимости и допустимости изменения. Это объясняется тем, что некоторые изменения вызваны не ошибками в программе, а неправильным пониманием требований, другие могут дублировать исправление ранее обнаруженных ошибок. Если в процессе проверки выявляется, что изменение недопустимо, повторяется или уже было рассмотрено, то изменение отклоняется. Лицу, представившему запрос на изменение, объясняется причина отказа.

Для принятых изменений начинается вторая стадия – оценка изменений и предварительное определение стоимости. Сначала следует проверить влияние изменения на всю систему. Для этого делается технический анализ способа внесения изменения. Затем определяется стоимость внесения изменения в определенные компоненты, что регистриру-

ется в форме запроса. В процессе оценивания полезна база данных конфигураций с информацией о взаимосвязях между компонентами, благодаря чему есть возможность оценить влияние изменений на другие компоненты системы.

Все изменения, кроме тех, которые относятся к исправлению мелких недоработок, должны быть переданы в группу контроля за изменениями, где принимается решение о принятии изменения либо отказе. Эта группа оценивает воздействие изменения не с технической, а скорее с организационной или стратегической точек зрения. Во внимание принимаются такие соображения, как экономическая выгодность изменения и организационные факторы, которые оправдывают необходимость изменения.

Группа контроля за изменениями состоит из лиц, на которых возлагается ответственность за решения о внесении изменений. Такие группы со структурой, включающей старшего менеджера компании-заказчика и сотрудников фирмы-разработчика, обязательны при выполнении военных проектов. Для небольших или среднего размера проектов в эту группу может входить только менеджер проекта и один-два инженера, которые не занимались разработкой данного ПО. В отдельных случаях допускается участие аналитика по изменениям, который дает рекомендации относительно того, оправданы эти изменения либо нет.

После принятия решения о внесении изменений программная система для внесения изменений передается разработчикам или команде по сопровождению системы. По окончании этой процедуры система обязательно должна пройти проверку на правильность внесения изменений. После этого именно команда по управлению конфигурацией, а не разработчики, займется выпуском новой версии.

Изменение каждого компонента системы должно регистрироваться. Таким образом создается история компонента. Самый лучший способ для этого – создавать стандартизированные комментарии в начале кода компонента (листинг 29.2), где содержатся ссылки на запросы изменений данного компонента. Для составления отчетов об изменениях компонента и обработки их историй используются специальные средства.

Листинг 29.2. Заголовок компóнента

```
// Проект PROTEUS (ESPRIT 6087)
//
// PCL-TOOLS/ПРАВКА/ФОРМЫ/ОТОБРАЖЕНИЕ/ИНТЕРФЕЙСЫ
//
// Объект: PCL-Tool-Desc
// Автор: Г.Дин
// Дата создания: 10 ноября 1998 г.
//
// © Lancaster University 1998
//
// История изменений
// Версия  Кто внес изменения Дата     Изменение  Причина
// 1.0.    Дж. Джонс      1/12/1998  Добавление Предложена
//                                   заголовка  группой по
//                                              управлению
//                                              конфигурацией
// 1.1.    Г. Дин         9/4/1999       Новое поле     Запрос R07/99
```

29.3. Управление версиями и выпусками

Управление версиями и выпусками ПО необходимо для идентификации и слежения за всеми версиями и выпусками системы. Менеджеры, отвечающие за управление версиями и выпусками ПО, разрабатывают процедуры поиска нужных версий системы и следят за

тем, чтобы изменения не осуществлялись произвольно. Они также работают с заказчиками и планируют время выпуска следующих версий системы. Над новыми версиями системы должна работать команда по управлению конфигурацией, а не разработчики, даже если новые версии предназначены только для внутреннего использования. Только в том случае, если информация об изменениях в версиях вносится исключительно командой по управлению конфигурацией, можно гарантировать согласованность версий.

Версией системы называют экземпляр системы, имеющий определенные отличия от других экземпляров этой же системы. Новые версии могут отличаться функциональными возможностями, эффективностью или исправлениями ошибок. Некоторые версии имеют одинаковую функциоиальность, однако разработаны под различные конфигурации аппаратного или программного обеспечения. Если отличия между версиями иезначительны, они называются *вариантами* одной версии.

Выходная версия (release) системы — это та версия, которая поставляется заказчику. В каждой выходной версии либо обязательно присутствуют новые функциональные возможности, либо она разработана под новую платформу. Количество версий обычно намного превышает количество выходных версий, поскольку версии создаются в основном для внутреннего пользования и не поставляются заказчику.

В настоящее время для поддержки управления версиями разработано много разнообразных CASE-средств (см. раздел 29.5). С помощью этих средств осуществляется управление хранением каждой версии и контроль за допуском к компонентам системы. Компоненты могут извлекаться из системы для внесения в них изменений. После введения в систему измененных компонентов получается новая́ версия, для которой с помощью системы управления версиями создается новое имя.

29.3.1. Идентификация версий

Любая большая программная система состоит из сотен компонентов, каждый из которых может иметь несколько версий. Процедуры управления версиями должны четко идентифицировать каждую версию компонента. Существует три основных способа идентификации версий.

1. *Нумерация версий.* Каждый компонент имеет уникальный и явный номер версии. Эта схема идентификации используется наиболее широко.
2. *Идентификация, основанная на значениях атрибутов.* Каждый компонент идентифицируется именем, которое, однако, не является уникальным для разных версий, и набором значений атрибутов, разных для каждой версии компонента [110]. Здесь версия компонента идентифицируется комбинацией имени и набора значений атрибутов.
3. *Идентификация на основе изменений.* Каждая версия системы именуется так же, как в способе идентификации, основанном на значениях атрибутов, плюс ссылки на запросы на изменения, которые реализованы в данной версии системы [244]. Таким образом, версия системы идентифицируется именем и теми изменениями, которые реализованы в системных компонентах.

Нумерация версий

По самой простой схеме нумерации версий к имени компонента или системы добавляется номер версии. Например, Solaris 2.6 обозначает версию 2.6 системы Solaris. Первая версия обычно обозначается 1.0, последующими версиями будут 1.1, 1.2 и т.д. На каком-то этапе создается новая выходная версия — версия 2.0, нумерация этой версии начинается заново — 2.1, 2.2 и т.д. Эта линейная схема нумерации основана на предположении о последовательности создания версий. Подобный подход к идентификации версий поддерживается многими программными средствами управления версиями, например RCS (см. раздел 29.5).

На рис. 29.3 графически проиллюстрирован описанный способ нумерации версий. Стрелки на рисунке проведены от исходной версии к новой, которая создается на ее основе. Отметим, что последовательность версий не обязательно линейная – версии с последовательными номерами могут создаваться на основе разных базовых версий. Например, на рис. 29.3 видно, что версия 2.2 создана на основе версии 1.2, а не версии 2.1. В принципе каждая существующая версия может служить основой для создания новой версии системы.

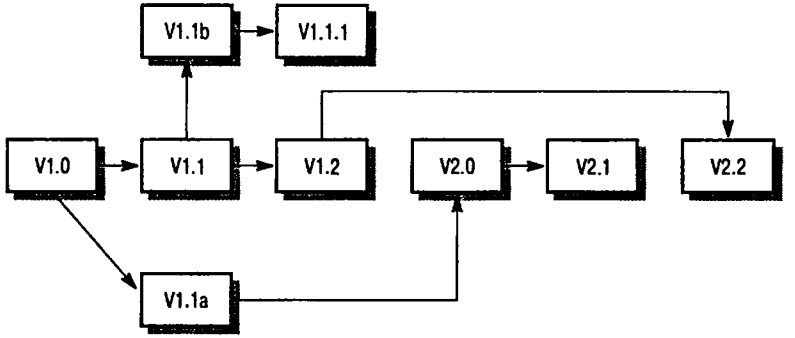


Рис. 29.3. Структура системных версий

Данная схема идентификации версий достаточно проста, однако она требует довольно большого количества информации для сопоставления версий, что позволяло бы отслеживать различия между версиями и связи между запросами на изменения и версиями. Поэтому поиск отдельной версии системы или компонента может быть достаточно трудным, особенно при отсутствии интеграции между базой данных конфигураций и системой хранения версий.

Идентификация, основанная на значениях атрибутов

Основная проблема схем явного именования версий заключается в том, что такие схемы не отображают тех признаков, которые можно использовать для идентификации версий, например:

- заказчик;
- язык программирования;
- состояние разработки;
- аппаратная платформа;
- дата создания.

Если каждая версия определяется единым набором атрибутов, нетрудно добавить новые версии, основанные на любой из существующих версий, поскольку они будут идентифицироваться единым набором значений атрибутов. При этом значения многих атрибутов новой версии будут совпадать со значениями атрибутов исходной версии; таким образом можно прослеживать взаимоотношения между версиями. Поиск версий осуществляется на основе значений атрибутов. При этом возможны такие запросы, как "самая последняя версия", "версия, созданная между определенными датами" и т.п. Например, обозначение для версии системы AC3D, разработанной на языке Java для использования под управлением Windows NT в январе 1999 года, будет выглядеть следующим образом:

AC3D (язык = Java, платформа = NT4, дата = январь 1999).

Идентификация, основанная на значениях атрибутов, системой управления версиями может применяться непосредственно. Однако более распространено использование только части имени версии, при этом база данных конфигураций поддерживает связь между значениями атрибутов и версиями системы и компонентов.

Идентификация на основе изменений

Идентификация, основанная на значениях атрибутов, устраняет проблему поиска версий, свойственную простым схемам нумерации, когда для поиска версии требуется знание ее атрибутов. Но в этом случае для регистрации взаимосвязей между версиями и изменениями необходимо использование отдельной системы управления изменениями.

Идентификация на основе изменений применяется скорее к системам, чем к системным компонентам; версии отдельных компонентов скрыты от пользователей системы управления конфигурацией. Каждое изменение в системе описывается *массивом изменений*, где указаны изменения в отдельных компонентах, реализующие данное системное изменение. Массивы изменений могут применяться последовательно таким образом, чтобы создать версию системы, в которой реализованы все необходимые изменения. В этом случае не требуется точного обозначения версии. Команда управления конфигурацией работает с системой управления версиями посредством системы управления изменениями.

Естественно, применение нескольких массивов изменений к системе должно быть согласовано, поскольку отдельные массивы изменений могут быть несовместимыми и их последовательное применение может привести к появлению неработоспособной системы. Кроме того, массивы изменений могут конфликтовать, если они предполагают разные изменения в одном компоненте. Для устранения этих проблем применяются средства управления версиями, поддерживающие идентификацию на основе изменений, что позволяет установить точные правила согласованности последовательности системных версий и что, в свою очередь, ограничивает способы комбинирования массивов изменений.

29.3.2. Управление выходными версиями

Выходной версией системы называется версия, поставляемая заказчику. Менеджеры по выпуску выходных версий отвечают за решение о дате выпуска, за управление процессом создания выходной версии, а также за создание документации.

Выходная версия системы включает в себя не только системный код, но также ряд компонентов.

1. *Конфигурационные файлы*, определяющие способ конфигурирования системы для каждой инсталляции.
2. *Файлы данных*, необходимые для работы системы.
3. *Программа установки*, которая помогает инсталлировать систему.
4. *Документация* в электронном и печатном виде, описывающая систему.
5. *Упаковка и рекламные материалы*, разработанные специально для этой версии системы.

Менеджеры по выпуску выходных версий не могут быть уверены, что заказчики всегда будут заменять старые версии системы новыми. Некоторые пользователи вполне удовлетворены установленными у них версиями и считают, что установка новых версий не стоит затрат. Поэтому новые выходные версии системы не должны зависеть от предыдущих. Рассмотрим следующую ситуацию.

1. Версия 1 системы находится в эксплуатации.
2. Выпускается версия 2, требующая установки новых файлов данных. Однако некоторые пользователи не нуждаются в дополнительных возможностях версии 2 и продолжают использовать версию 1.
3. Версия 3 требует файлов, содержащихся в версии 2, но сама не содержит этих файлов.

Дистрибьютор ПО не может знать наверняка, что файлы данных, требующиеся для версии 3, уже установлены; некоторые пользователи будут переходить от версии 1 к версии 3, минуя версию 2. У других пользователей вследствие каких-либо обстоятельств файлы данных, связанные с версией 2, могут быть изменены. Отсюда следует простой вывод: версия 3 должна содержать все файлы данных.

Принятие решения о выпуске выходной версии

Подготовка и распространение программных систем требуют больших затрат, особенно это касается рынка массовых программных продуктов. Если выпуски выходных версий осуществляются слишком часто, пользователи не успеют осознать потребность в расширенных возможностях новых версий, а если выходные версии создаются редко, существует вероятность потери рынка сбыта, поскольку пользователи переходят к альтернативным системам. Это не относится к программным продуктам, созданным под заказ для определенной организации. Однако и тут редкие выходные версии могут привести к расхождению программной системы и тех бизнес-процессов, для поддержки которых система была разработана.

Принятие решения о том, когда именно должна выйти следующая выходная версия системы, существенно зависит от технических и общих организационных факторов, которые описаны в табл. 29.1.

Таблица 29.1. Факторы, влияющие на стратегию выпуска версий системы

Фактор	Описание
Техническое качество системы	Необходимость выпуска новой версии обусловлена зарегистрированными ошибками в существующей версии системы. Небольшие дефекты можно устранить с помощью заплат (patches), которые часто распространяются через Internet
Пятый закон Лемана (см. главу 27)	Этот закон постулирует постоянство приращения функциональных возможностей в каждой выходной версии по сравнению с предыдущей. Однако существуют и исключения, например за версией с достаточно большими изменениями следует версия с исправлением ошибок
Конкуренция	Необходимость новой версии объясняется наличием на рынке конкурирующих продуктов
Требования рынка	Отдел маркетинга компании может приурочить выход новой версии к определенной дате
Предложения заказчика об изменениях в системе	Для разработанных под заказ систем заказчик может предложить внести в систему ряд изменений, тогда новая версия выйдет сразу после реализации этих изменений

Создание выходной версии

Создание выходной версии — это процесс сбора всех необходимых файлов и документации, составляющих выходную версию системы. Требуется определить нужные исполняемые коды программ и файлы с данными. Конфигурация выходной версии должна определяться под конкретный тип аппаратных средств и операционной системы. Также нужно подготовить инструкции для пользователей по инсталляции системы, в том числе

в электронном виде. Должны быть написаны сценарии для инсталляционной программы. В завершение создается инсталляционный диск, на котором будет распространяться система. В настоящее время в качестве носителей дистрибутивов наиболее широко распространены компакт-диски емкостью до 600 Мбайт.

Документирование выходной версии

Процесс создания выходной версии должен быть задокументирован, чтобы была возможность восстановить ее в будущем. Это особенно важно для больших систем с длинным жизненным циклом, разрабатываемых под заказ. Заказчики обычно используют одну версию системы на протяжении многих лет, и все необходимые изменения вносятся именно в эту версию через много лет после ее поставки.

Для документирования выходной версии прежде всего необходимо записать версии исходного кода компонентов, которые использованы для создания исполняемого кода. Также следует собрать и сохранить все копии исходных и исполняемых кодов, системных данных и конфигурационных файлов. Кроме того, должны быть записаны версии операционной системы, библиотеки, компиляторы и другие средства, применяемые для сборки системы.

29.4. Сборка системы

Сборкой системы называют процесс компиляции и связывания программных компонентов в единую исполняемую программу. Перед сборкой системы полезно ответить на следующие вопросы.

1. Все ли компоненты, составляющие систему, включены в инструкцию по сборке?
2. Каковы версии компонента, перечисленные в инструкции по сборке?
3. Доступны ли все необходимые файлы данных?
4. Если на файлы данных используются ссылки внутри компонентов, то каковы имена этих файлов в выходной версии?
5. Доступны ли нужные версии компилятора и других необходимых средств? Действующие версии программных средств могут быть несовместимы с более старыми версиями, которые применялись при разработке системы.

В настоящее время существует много средств управления конфигурацией, автоматизирующих процесс сборки системы. Команда управления конфигурацией пишет сценарий, в котором определены зависимости между различными компонентами системы. В нем также указаны средства компилирования и связывания компонентов системы. Средства компоновки интерпретируют сценарий сборки системы и вызывают программы, необходимые для сборки исполняемой системы. Процесс сборки системы представлен на рис. 29.4.

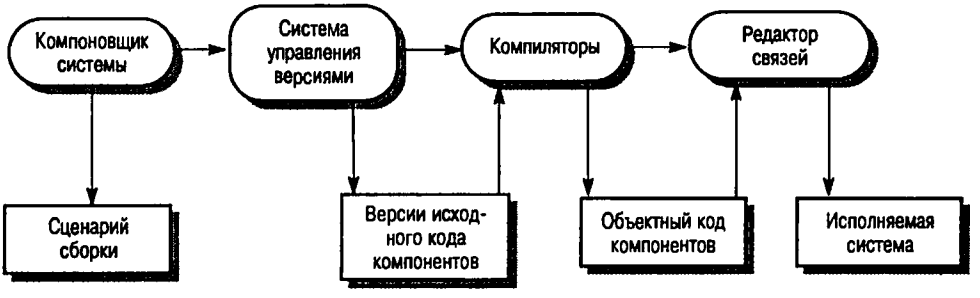


Рис. 29.4. Сборка системы

В сценарии сборки указаны зависимости между компонентами, поэтому компоновщик системы сам принимает решение, когда перекомпилировать компоненты, а когда можно многократно использовать существующий объектный код. Зависимости в сценарии сборки указаны в основном как зависимости между файлами, содержащими исходный код компонентов. Однако, если файлов с исходным кодом разных версий много, возникает проблема выбора нужных файлов. Проблема усугубляется, если файлы исходного и объектного кода имеют одинаковые имена (но, конечно, с разными расширениями).

Чтобы избежать трудностей, связанных с зависимостью физических файлов, было разработано несколько экспериментальных систем, основанных на языках описания модулей [320]. В них используется описание логической структуры ПО и схемы зависимостей между файлами, содержащими компоненты исходного кода. Такой подход снижает количество ошибок и приводит к более понятным описаниям процесса сборки системы.

29.5. CASE-средства для управления конфигурацией

Процесс управления конфигурацией обычно стандартизирован и включает выполнение заранее определенных процедур. Они требуют детализированного контроля за очень большим количеством данных. При сборке системы единственная ошибка в управлении может привести к некорректной работе системы. Поэтому очень важна поддержка процесса управления конфигурацией соответствующими CASE-средствами. Начиная с 70-х годов было разработано большое количество программных средств поддержки разных аспектов процесса управления конфигурацией.

Примерами первого поколения средств управления конфигурацией могут служить системы SCCS [297] и RCS [335], предназначенные для управления версиями и сборкой систем [114]. Это автономные средства, которые поддерживали отдельные действия в процессе управления конфигурацией. Средства второго поколения, например Lifespan [342] и DSEE [212], обеспечивают интегрированную поддержку процесса управления конфигурацией, однако некоторые этапы управления они не обеспечивали. Во время написания данной книги были доступны интегрированные пакеты CASE-средств, поддерживающие планирование управления, процессы управления изменениями, версиями и сборкой системы [211]. Однако эти пакеты достаточно сложные, требуют усилий для изучения и освоения, поэтому многие организации-разработчики продолжают использовать средства поддержки первого и второго поколений[1].

29.5.1. Средства поддержки управления изменениями

Процесс управления изменениями заключается в заполнении форм запросов на изменения, проведении анализа изменений и передаче этих форм и соответствующих конфигурационных элементов команде управления качеством и команде по управлению конфигурацией. Этот алгоритмический по своей природе процесс позволяет сравнительно легко интегрировать его с системой управления версиями, поскольку, упрощая, можно сказать, что задача управления изменениями заключается в передаче нужных документов нужным людям в нужное время.

---

[1] *Кроме перечисленных CASE-средств поддержки процесса управления конфигурацией, назовем еще Rational ClearCase от Rational Software и семейство продуктов PVCS Professional и PVCS Notify от фирмы Merant, которые можно встретить у многих российских разработчиков ПО.* – Прим. ред.

Поэтому для поддержки процесса управления изменениями достаточно следующих средств.

1. *Редактор форм*, позволяющий создавать и заполнять формы запросов на изменения.
2. *Система автоматизации документооборота*, которая позволяет фиксировать закрепление обработки форм запросов на изменения за членами команды по управлению конфигурацией и определяет порядок этой обработки. Эта система может также автоматизировать процесс передачи заполненных форм "нужным людям в нужное время" и информировать о состоянии процесса внесения изменений. Как правило, эта система использует электронную почту для пересылки сообщений.
3. *База данных изменений*, которая используется для хранения всех предложенных изменений и может быть связана с системой управления версиями.

29.5.2. Средства поддержки управления версиями

Управление версиями предполагает обработку больших массивов информации для регистрации изменений, вносимых в систему, и контроля за ними. Средства управления версиями обязательно включают репозиторий конфигурационных элементов, которые в дальнейшем не изменяются. Если необходимо изменить какой-либо конфигурационный элемент, находящийся в репозитории, его копия помещается в рабочий каталог. После изменений новая версия элемента также помещается в репозиторий.

Системы управления версиями могут отличаться друг от друга, но все они имеют базовый набор средств.

1. *Средство идентификации версий.* Системы управления версиями могут поддерживать различные подходы к идентификации версий (см. раздел 29.3.1).
2. *Средство управления хранением версий.* Чтобы уменьшить пространство, необходимое для хранения различных версий системы, которые могут быть значительных размеров, системы управления версиями используют специальные средства управления хранением, когда хранятся не сами версии, а их отличия от некоторой базовой версии. Различия между версиями представляются в виде *дельты*, где собраны инструкции, необходимые для воссоздания соответствующей версии системы. На рис. 29.5 показано, как из последней версии можно восстановить более раннюю версию системы.

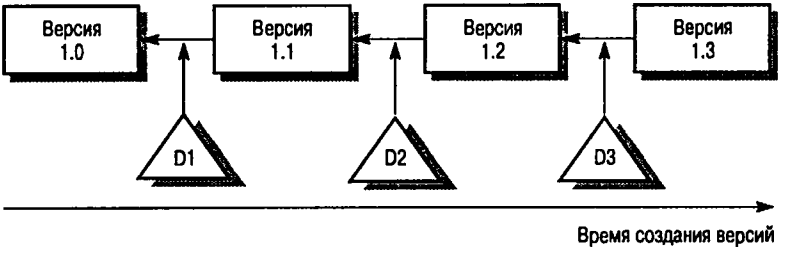


Рис. 29.5. Восстановление версий

3. *Средство регистрации изменений.* Регистрирует все изменения, сделанные в коде системных компонентов. В некоторых системах управления версиями это средство используется для поиска нужной версии системы.
4. *Средство поддержки параллельной разработки.* Различные версии системы могут разрабатываться параллельно и изменяться независимо друг от друга. Система управле-

ния версиями должна отслеживать компоненты, которые изменяются, и контролировать, чтобы на один и тот же компонент не накладывались изменения, сделанные разными группами разработчиков. Некоторые системы позволяют единовременно изменять только один экземпляр компонента, другие автоматически разрешают возникшие коллизии, когда измененные компоненты возвращаются в систему управления версиями.

29.5.3. Средства сборки систем

Сборка систем – это очень трудоемкий вычислительный процесс. Например, процесс компиляции большой системы, состоящей из сотен компонентов, может занять несколько часов. Если компиляцию и связывание компонентов такой системы выполнять вручную, то оператор неизбежно сделает какие-либо ошибки. Средства сборки систем автоматизируют этот процесс, что исключает потенциальные ошибки, совершаемые при ручном компилировании, и, возможно, сокращает время сборки системы.

Средства сборки систем могут быть как автономными, например соответствующие утилиты в системе Unix [114], так и интегрированными со средствами управления версиями. Как правило, CASE-средства сборки систем состоят из следующих компонентов.

1. *Язык спецификирования зависимостей и соответствующий интерпретатор.* Описывает и управляет зависимостями между системными компонентами и минимизирует возможные перекомпиляции.
2. *Средства выбора и реализации.* Это компиляторы и другие средства работы с файлами исходного кода.
3. *Средства распределенной компиляции.* Некоторые компоновщики систем, особенно интегрированные с системами управления конфигурациями, могут поддерживать распределенную (сетевую) компиляцию. Вместо выполнения всего процесса компиляции на одной машине компоновщик находит свободные процессоры в компьютерной сети и организует параллельную компиляцию. Это значительно сокращает время сборки системы.
4. *Средство управления вторичными объектами.* Вторичные – это объекты, которые создаются на основе других, исходных, объектов. Средство управления такими объектами связывает исходный код и вторичные объекты и создает новые объекты только тогда, когда изменяется исходный код.

Управление вторичными объектами и минимизацию количества перекомпиляций лучше всего объяснить на простом примере. Рассмотрим ситуацию, когда программа comp создается из объектных модулей scan.o, syn.o, sem.o и cgen.o. Для объектных модулей существуют модули, содержащие исходный код и имеющие имена соответственно scan.c, syn.c, sem.c и cgen.c. Эти модули используют общий файл объявлений defs.h. Зависимости между модулями показаны стрелками на рис. 29.6 (стрелки можно "прочитать" как "зависит от").

Допустим, в модуль scan.c внесены изменения. Система сборки должна определить, что вторичный объект scan.o необходимо создать заново, и вызвать соответствующий компилятор для перекомпиляции модуля scan.c и создания нового экземпляра объекта scan.o. Далее система сборки на основании связи между comp и scan.o определяет, что необходимо заново создать также программу comp путем связывания модулей scan.o, syn.o, sem.o и cgen.o. При этом система определяет, что объектный код других компонентов не изменялся, поэтому перекомпиляция их исходного кода не требуется.

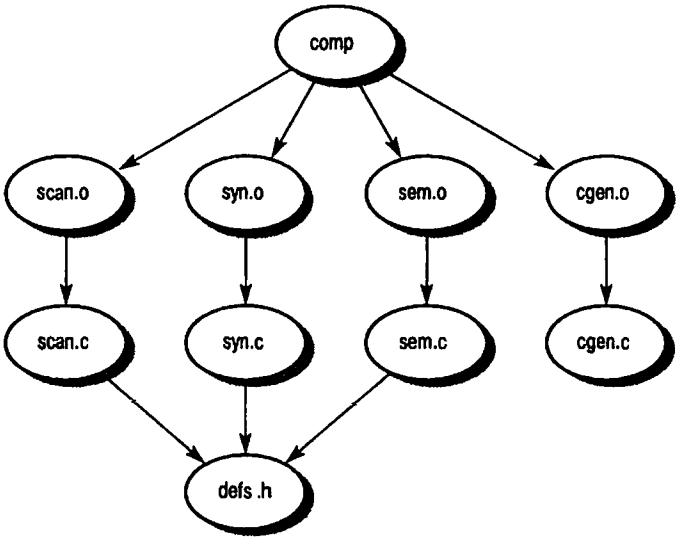


Рис. 29.6. Схема зависимостей модулей

Некоторые системы сборки используют дату изменения файла как ключевой атрибут, определяющий, требуется или нет перекомпиляция. Если дата изменения файла исходного кода более поздняя, чем дата изменения соответствующего файла объектного кода, то этот объектный код необходимо создать заново. Это гарантирует, что вторичный объект будет создан на основе самой последней версии исходного кода. Если перекомпилируется ранняя версия исходного кода, то изменяется дата ее модификации и система сборки по этой дате определяет, какие компоненты должны быть перекомпилированы или созданы заново. Другие системы сборки используют более сложные подходы к управлению вторичными объектами. Они вводят дополнительный атрибут для вторичных объектов, где указывается версия исходного кода, на основе которого создан этот объект, и по возможности сохраняются все версии вторичных объектов. Это позволяет иметь объектный код всех версий исходного кода без дополнительной перекомпиляции.

КЛЮЧЕВЫЕ ПОНЯТИЯ

- Управление конфигурацией — это управление изменениями в программной системе. Основная роль команды по управлению конфигурацией заключается в правильном внесении изменений в систему.
- В больших проектах для отслеживания различных версий всех проектных документов разрабатывается и используется формальная схема именования документов.
- Для управления конфигурацией необходима база данных конфигураций, где хранится вся информация о проведенных в системе изменениях и запросы на изменения.
- Для управления конфигурацией необходима схема идентификации версий. Версии можно идентифицировать по номерам, на основе значений атрибутов и на основе изменений, внесенных в систему.
- Выходная версия системы включает исполняемый код, файлы данных, конфигурационные файлы и документацию. Управление выходными версиями предусматривает определение даты выпуска системы, подготовку всей информации для распространения системы и документирование каждой выходной версии.

- Сборка системы — это процесс компоновки системных компонентов в виде исполняемой программы.
- Для поддержки процесса управления конфигурацией применяются CASE-средства. Они включают средства для управления версиями и изменениями и средства для сборки системы.

Упражнения

29.1. Объясните, почему в системе управления конфигурацией для идентификации документов не используются названия документов. Предложите схему идентификации документов, которую можно было бы использовать во всех проектах вашей организации.

29.2. Используя модель "сущность–связь" или объектно-ориентированный подход (см. главу 7), разработайте модель базы данных конфигураций, которая должна содержать информацию о системных компонентах, их версиях, выходных версиях системы и об изменениях, реализованных в системе. База данных должна обладать следующими функциями:

- извлечение всех версий или отдельной указанной версии компонента;
- извлечение последней по времени изменения версии компонента;
- поиск запросов на изменения, которые были реализованы в указанной версии системы;
- определение версий компонентов, включенных в указанную версию системы;
- извлечение выходной версии системы, определяемой по дате выпуска или по имени заказчика, которому она поставлена.

29.3. С помощью диаграммы потока данных представьте модель управления изменениями, которую можно было бы применить в большой организации, занимающейся разработкой ПО для внешних заказчиков. Запросы на изменения могут поступать как от внешних, так и от внутренних источников.

29.4. Опишите трудности, которые могут встретиться при сборке системы. В частности, рассмотрите проблемы сборки системы на хост-компьютере.

29.5. Со ссылкой на систему сборки объясните, почему иногда необходимо сохранять устаревшие компьютеры, на которых разрабатывались большие программные системы.

29.6. При сборке систем часто возникает сложная проблема, состоящая в том, что имена физических файлов встроены в системный код и используемая файловая структура отличается от файловой структуры на конечной машине, где будет установлена система. Составьте руководство для программистов, которое поможет избежать этой и подобных проблем при сборке систем.

29.7. Приведите пять факторов, которые необходимо учитывать при сборке выходных версий больших программных систем.

29.8. Опишите два способа оптимизации процесса сборки системы из ее компонентов с помощью соответствующих CASE-средств компоновки систем.

Литература

1. Abbott R. Program design by informal English descriptions // *Comm. ACM.* – 1983. – **26** (11). – P. 882–976.
2. Abdel-Ghaly A. A., Chan P. Y. *et al.* Evalution of competing software reliability predictions // *IEEE Trans. on Software Engineering.* – 1986. – **SE-12** (9). – P. 950–1017.
3. Ackroyd S., Harper R. *et al. Information Technology and Practical Police Work.* – Milton Keynes : Open University Press, 1992.
4. Adams E. N. Optimizing preventative service of software products // *IBM J. Res & Dev.* – 1984. – **28** (1). – P. 2–16.
5. Aho A. V., Kernighan B. W. *et al. The Awk Programming Language.* – Englewood Cliffs, NJ : Prentice-Hall, 1988.
6. Albrecht A. J. Measuring application development productivity // *SHARE/GUIDE IBM Application Development Symposium*, 1979.
7. Albrecht A. J. and Gaffney J. E. Software function, lines of code and development effort prediction: a software science validation // *IEEE Trans. on Software Engineering.* – 1983. – **SE-9** (6). – P. 639–686.
8. Alexander C., Ishikawa S. *at al. A Pattern Language.* – Oxford : Oxford University Press, 1977.
9. Alford M. W. A requirements engineering methodology for real time processing requirements // *IEEE Trans. on Software Engineering.* – 1977. – **SE-3** (1). – P. 60–69.
10. Alford M. W. SREM at the age of eight: the distributed computing design system // *IEEE Computer.* – 1985. – **18** (4). – P. 36–82.
11. Aron J. D. *The Program Development Process.* – Reading, MA : Addison-Wesley, 1974.
12. Aron J. D. *The Program Development Process: Part 2 - The Programming Team.* – Reading, MA : Addison-Wesley, 1983.
13. Arthur L. J. *Software Evolution.* – New York : John Wiley and Sons, 1988.
14. Avizienis A. The N-Version Approach to Fault-Tolerant Software // *IEEE Trans. on Software Engineering.* – 1985. – **SE-11**(12). – P. 1491–1992.
15. Avizienis A. A. A methodology of N-version programming. – In: *Software Fault Tolerance* (M. R. Lyu, ed.). – Chichester : John Wiley and Sons, 1995. – P. 23–69.
16. Baker F. T. Chief programmer team management of production programming // *IBM Systems J.* – 1972. – **11** (1). – P. 56–129.
17. Baker T. P. and Scallon G. M. An architecture for real-time software systems // *IEEE Software.* – 1986. – **3** (3). – P. 50–58.
18. Banker R. D., Datar S. M. *et al.* Software complexity and maintenance costs // *Comm. ACM.* – 1993. – **36** (11). – P. 81–175.
19. Banker R., Kauffman R. *et al.* An empirical test of object-based output measurement metrics in a computer-aided software engineering (CASE) environment // *J. of Management Info. Sys.* – 1992. – **8** (3). – P. 127–177.
20. Bansler J. P. and Bodker K. A reappraisal of structured analysis: design in an organizational context // *ACM Trans. on Information Systers.* – 1993. – **11** (2). – P. 165–258.
21. Barker R. *CASE* Method: Entity Relationship Modelling.* – Wokingham : AddisoModelling ; Addison-Wesley, 1989.
22. Barnard J. and Price A. Managing code inspection information // *IEEE Software.* – 1994. – **11**(2). – P. 59–128.
23. Barnes J. *Programming in Ada 95.* – Harlow, Essex : Addison-Wesley, 1998.
24. Barnes J. G. P. *Programming in Ada, 4th edn.* – Wokingham : Addison-Wesley, 1994.

25. Basili V. and Green S. Software process improvement at the SEL // *IEEE Software.* – 1993. – **11**(4). – P. 58–124.
26. Basili V. R. and Rombach H. D. The TAME project: towards improvement-oriented software environments // *IEEE Trans. on Software Engineering.* – 1988. – **14**(6). – P. 758–1531.
27. Basili V. R. and Selby R. W. Comparing the effectiveness of software testing strategies // *Ibid.* – 1987. – **SE-13**(12). – P. 1278–1374.
28. Bass B. M. and Daunteman G. Bahaviour in groups as a function of self, interaction and task orientation // *J. abnorm. soc. Psychology.* – 1963. – 66(4). – P. 19–47.
29. Bass L., Clements P., *et al. Software Architecture in Practice.* – Reading, MA : Addison-Wesley, 1998.
30. Baumer D., Gryczan G. *et al.* Framework development for large systems // *Comm. ACM.* – 1997. – **40**(10). – P. 52–61.
31. Beck K. Embracing change with extreme programming // *IEEE Computer.* – 1999. – **32**(10). – P. 70–78.
32. Beck K. *Extreme Programming Explained.* – Reading, MA : Addison-Wesley Longman, 2000.
33. Beck K. and Cunningham W. A laboratory for teaching object-oriented thinking // *SIGPLAN Notices.* – 1989. – **24**(10).
34. Beizer B. *Software Testing Techniques, 2nd ed.* – New York : Van Nostrand Rheinhold, 1990.
35. Bell T. E., Bixler D. C. *et al.* An extendable approach to computer aided software requirements engineering // *IEEE Trans. on Software Engineering.* – 1977. – **SE-3**(1). – P. 49–109.
36. Bentley R., Rodden T. *et al.* Ethnographically-informed systems design for air traffic control. – In : *Proc. CSCW'92.* – Toronto, Canada, 1992.
37. Bernstein P. A. Middleware: a model for distributed system services // *Comm. ACM.* – 1996. – **39**(2). – P. 86–183.
38. Bersoff E. H. and Davis A. M. Impact of life cycle models on software configuration management // *Ibid.* – 1991. – **34**(8). – P. 104–122.
39. Binder R. V. *Testing Object-Oriented Systems: Models, Patterns and Tools.* – Reading, MA : Addison-Wesley Longman, 1999.
40. Bishop P., Esp D. *et al.* PODS – a project on diverse software // *IEEE Trans. on Software Engineering.* – 1986. – **SE-12**(9). – P. 929–969.
41. Boehm B. *COCOMO II Model Definition Manual.* – Computer Science Dept., University of Southern California, 1997.
42. Boehm B. COTS integration: plug and pray? // *IEEE Computer.* – 1999. – **32**(1). – P. 135–173.
43. Boehm B. and Royce W. Ada COCOMO and the Ada Process Model // *Proc. 5th COCOMO Users' Group Meeting.* – Pittsburgh, Software Engineering Institute, 1989.
44. Boehm B., Clark B. *et al.* Cost models for future life cycle proctsses: COCOMO 2 // *Annals of Software Engineering.* – 1995. – **1**. – P. 57–151.
45. Boehm B. W. Software engineering: R & D trends and defense needs. – In : *Research. Directions in Software Technology* (P. Wegner, ed.). – Cambridge, MA : MIT Press, 1979.
46. Boehm B. W. *Software Engineering Economics.* – Englewood Cliffs, NJ : Prentice-Hall, 1981. (Русский перевод: Боэм Б. У. Инженерное проектирование программного обеспечения. – М. : Радио и связь, 1985.)
47. Boehm B. W. A spiral model of software development and enhancement // *IEEE Computer.* – 1988. – **21**(5). – P. 61–133.
48. Boehm B. W., Gray T. E. *et al.* Prototyping versus specifying: A multi-project experiment // *IEEE Transactions on Software Engineering.* – 1984. – **SE-10**(3). – P. 290–593.
49. Boehm B. W., McClean R. L. *et al.* Some experience with automated aids to the design of large-scale reliable software // *Ibid.* – 1975. – **SE-1**(1). – P. 125–158.
50. Bohm C. and Jacopini G. Flow diagrams, Turing machines and languages with only two formation rules // *Comm. ACM.* – 1966. – **9**(5). – P. 366–437.

51. Bollinger T. and McGowan C. A critical look at software capability evaluations // *IEEE Software.* – 1991. – 8(4). – P. 25–66.
52. Bolognesi T. and Brinksma E. Introduction to the ISO specification language LOTOS // *Computer Networks.* – 1987. – 14(1). – P. 25–84.
53. Booch G. *Software Components with Ada: Structures Tools and Subsystems.* – Menlo Park, CA : Benjamin Cummings, 1987.
54. Booch G. *Object-oriented Analysis and Design with Applications.* – Redwood City, CA : Benjamin Cummings, 1994. (Русский перевод 2-го издания: Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд. – М. : Бином ; СПб. : Невский проспект, 1999.)
55. Booch G., Rumbaugh J. *et al. The Unified Modeling Language User Guide.* – Reading, MA : Addison-Wesley Longman, 1999. (Русский перевод: Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. – М. : ДМК, 2000.)
56. Bourne S. R. The Unix Shell // *Bell Sys. Tech. J.* – 1978. – 57(6). – P. 1971–2061.
57. Brazendale J. and Bell R. Safety-related control and protection systems: standards update // *IEEE Computing and Control Engineering J.* – 1994. – 5 (1). – P. 6–18.
58. Brilliant S. S., Knight J. C. *et al.* Analysis of faults in an N-version software experiment // *IEEE Trans. on Software Engineering.* – 1990. – 16 (2). – P. 238–285.
59. Brinch-Hansen P. *Operating System Principles.* – Englewood Cliffs, NJ : Prentice-Hall, 1973.
60. Brooks F. P. *The Mythical Man Month.* – Reading, MA : Addison-Wesley, 1975. (Русский перевод: Брукс П. Мифические человеко-месяцы. – М. : Мир, 1976. Есть также русский перевод последнего английского издания: Брукс П. Мифический человеко-месяц или как создаются программные системы. – СПб. : Символ-Плюс, 1999.)
61. Brown A. W., Earl A. N. *et al. Software Engineering Environments.* – London : McGraw-Hill, 1992.
62. Burns A. Scheduling hard real-time systems: a review // *BCS/IEE Software Engineering J.* – 1991. – 6(3). – P. 116–244.
63. Burns A. and Wellings A. *Real-time Systems and their Programming Languages.* – Wokingham : Addison-Wesley, 1990.
64. Burns A. and Wellings A. *Real-time Systems and Programming Languages, 2nd ed.* – Harlow, UK : Addison-Wesley, 1997.
65. Button G. and Sharrock W. The production of order and the order of production // *Proc. ECSCW'97.* – Lancaster, UK : Kluwer, 1997.
66. Buxton J. *Requirements for Ada Programming Support Environments: Stoneman.* – Washington D.C : US Department of Defense, 1980.
67. Card S., Moran T. P. *et al. The Psychology of Human-Computer Interaction.* – Hillsdale, NJ : Lawrence Erlbaum Associates, 1983.
68. Carroll J. M. *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill.* – Boston MA : MIT Press, 1992.
69. Checkland P. *Systems Thinking, Systems Practice.* – Chichester : John Wiley and Sons, 1981.
70. Checkland P. and Scholes J. *Soft Systems Methodology in Action.* – Chichester : John Wiley and Sons, 1990.
71. Chen P. The entity relationship model – Towards a unified view of data // *ACM Trans. on Database Systems.* – 1976. – 1(1). – P. 9–45.
72. Chikofsky E. J. and Cross J. H. Reverse engineering and design recovery: a taxonomy // *IEEE Software.* – 1990. – 7(1). – P. 13–30.
73. Cleveland L. A program understanding support environment // *IBM Sys. J.* – 1989. – 28(2). – P. 324–368.
74. Coad P. and Yourdon E. *Object-oriented Analysis.* – Englewood Cliffs, NJ : Prentice-Hall, 1990.

75. Cobb R. H. and Mills H. D. Engineering software under statistical quality control // *IEEE Software.* – 1990. – **7**(6). – P. 44–98.
76. Codd E. F. Extending the database relational model to capture more meaning // *ACM Trans. on Database Systems.* – 1979. – **4**(4). – P. 397–831.
77. Codenie W., De Hondt K. *et al.* From custom applications to domain-specific frameworks // *Comm. ACM.* – 1997. – **40**(10). – P. 70–77.
78. Cohen B., Harwood W. T. *et al. The Specification of Complex Systems.* – Wokingham : Addison-Wesley, 1986.
79. Constantine, L. L. and Yourdon, E. *Structured Design.* – Englewood Cliffs, NJ : Prentice-Hall, 1979.
80. Cooling, J. E. *Software Design for Real-time Systems.* – London : Chapman and Hall, 1991.
81. Coulouris G., Dollimore J. *et al. Distributed Systems: Concepts and Design.* – Wokingham : Addison-Wesley, 1994.
82. Crosby P. *Quality is Free.* – New York : McGraw-Hill, 1979.
83. Curtis B., Hefley W. E. *et al. Overview of the People Capability Maturity Model.* – Software Engineering Institute, 1995.
84. Curtis B., Krasner H. *et al.* A field study of the software design process for large systems // *Comm. ACM.* – 1988. – **31**(11). – P. 1268–1355.
85. Cusamano M. The software factory: a historical interpretation // *IEEE Software.* – 1989. – **6**(2). – P. 23–53.
86. Dasarthy B. Timing constraints for real-time systems: constructs for expressing them, methods of validating them // *IEEE Trans. on Software Engineering.* – 1985. – **SE-11** (1). – P. 80–86.
87. Date C. J. and Darwen H. *A Guide to the SQL Standard, 4th ed.* – Reading, MA : Addison-Wesley, 1997.
88. Davis A. M. *Software Requirements: Analysis and Specification.* – Englewood Cliffs, NJ : Prentice-Hall, 1990.
89. Davis A. M. *Software Requirements: Objects, Functions and States.* – Englewood Cliffs, NJ : Prentice-Hall, 1993.
90. Dehbonei B. and Mejia F. Formal development of safety-critical software systems in railway signalling. In : *Applications of Formal Methods* (M. Hinchey and J. P. Bowen, eds). – London : Prentice-Hall, 1995. – P. 227–279.
91. DeMarco T. *Structured Analysis and System Specification.* – New York : Yourdon Press, 1978.
92. DeMarco T. and Lister T. Programmer performance and the effects of the workplace. – In : *Proc. 8th Int. Conf. on Software Engineering.* – London : IEEE Press, 1985.
93. DeMarco T. and Lister T. *Peopleware: Productive Projects and Teams.* – New York : Dorset House, 1999.
94. Diaper D. *Task Analysis for Human-Computer Interaction.* – Chichester : Ellis Horwood, 1989.
95. Diaz M. and Sligo J. How software process improvement helped Motorola // *IEEE Software.* – 1997. – **14**(5). – P. 75–157.
96. Dijkstra E. W. Goto statement considered harmful // *Comm. ACM.* – 1968a. – **11** (3). – P. 147–155.
97. Dijkstra E. W. Cooperating sequential processes. – In : *Programming Languages* (F. Genuys, ed.). – London : Academic Press, 1968b. – P. 43–155.
98. Diller A. *Z: An Introduction to Formal Methods, 2nd ed.* – New York : John Wiley and Sons, 1994.
99. Dix A., Finlay J. *et al. Human-Computer Interaction, 2nd ed.* – London : Prentice-Hall, 1998.
100. Easterbrook S. Domain modelling with hierarchies of alternative viewpoints // *Proc. RE'93.* – San Diego, USA, 1993.
101. Easterbrook S. and Nuseibeh B. Using viewpoints for inconsistency management // *BCS/IEE Software Engineering J.* – 1996. – **11**(1). – P. 31–74.
102. Easterbrook S., Lutz R. *et al.* Experiences using lightweight formal methods for requirements modeling // *IEEE Trans. on Software Engineering.* – 1998. – **24**(1). – P. 4–18.
103. Eckstein R., Loy M. *et al. Java Swing.* – Sebastopol, CA : O'Reilly and Associates Inc., 1998.

104. ECMA *A Reference Model for Frameworks of Computer-Assisted Software Engineering Environments.* – Tech. Rept. European Comtuter Manufacturers Association, 1991.
105. Ehrlich W., Prasanna B. et al. Determining the cost of a stop-test decision // *IEEE Software.* – 1993. – **9**(4). – P. 33–75.
106. El Amam K., Drouin J. *et al. SPICE: The Theory and Practice of Software Process Improvement and Capability Determination.* – Los Alamitos, CA : IEEE Computer Society Press, 1997.
107. Ellison R. J., Fisher D. A. *et al.* Survivability: protecting your critical systems // *IEEE Internet Computing.* – 1999a. – **3**(6). – P. 55–118.
108. Ellison R. J., Linger R. C. *et al.* Survivable network system analysis: a case study // *IEEE Software.* – 1999b. – **16**(4). – P. 70–77.
109. Endres A. An analysis of errors and their causes in system programs // *IEEE Trans. on Software Engineering.* – 1975. – **SE-1**(2). – P. 140–149.
110. Estublier J. and Casallas R. The Adele Configuration Manager. – In : *Configuration Management* (W. Tichy, ed.). – Chichester : John Wiley and Sons, 1994. – P. 99–233.
111. Fagan M. E. Design and code inspections to reduce errors in program development // *IBM Systems J.* – 1976. – **15**(3). – P. 182–393.
112. Fagan M. E. Advances in software inspections // *IEEE Trans. on Software Engineering.* – 1986. – **SE-12**(7). – P. 744–795.
113. Fayad M. E. and Schmidt D. C. Object-oriented application frameworks // *Comm. ACM.* – 1997. – **40**(10). – P. 32–70.
114. Feldman S. I. MAKE – a program for maintaining computer programs // *Software-Practice and Experience.* – 1979. – **9**(4). – P. 255–320.
115. Finkelstein A., Kramer J. *et al.* Viewpoint oriented software development. – In : *Proc. 3rd Int. Workshop on Software Engineering and its Applications.* – Toulouse, France, 1990.
116. Forte G. Tools fair: out of the lab, onto the shelf // *IEEE Software.* – 1992. – **9**(3). – P. 70–79.
117. Fowler M. and Scott K. *UML Distilled: Applying the Standard Object Modelling Language.* – Reading, MA : Addison-Wesley, 1997.
118. Frewin G. D. and Hatton B. J. Quality management – procedures and practises // *IEE/BCS Software Engineering J.* – 1986. – **1**(1). – P. 29–67.
119. Fromme B. and Walker J. An open architecture for tool and process integration. – In : *Proc. 6th Conf. on Software Engineering Environments.* – Reading, UK : IEEE Press, 1993.
120. Fuggetta A. A classification of CASE technology // *IEEE Computer.* – 1993. – **26**(12). – P. 25–63.
121. Fujiwara E. and Pradhan D. K. Error-control coding in computers // *IEEE Computer.* – 1990. – **23**(7). – P. 63–135.
122. Furey S. and Kitchenham B. Point/Counterpoint: function points // *IEEE Software.* – 1997. – **14**(2). – P. 28–59.
123. Futatsugi K., Goguen J. A. *et al.* Principles of OBJ2. – In : *12th ACM Symp. on Principles of Programming Languages.* – New Orleans, 1985.
124. Gamma E., Helm R. *et al. Design Patterns: Elements of Reusable Object-Oriented Software.* – Reading, MA : Addison-Wesley, 1995. (Русский перевод: Гамма Э., Хелм Р., Джонсон Р., Влассидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб. : Питер, 2001.)
125. Gane C. and Sarson T. *Structured Systems Analysis.* – Englewood Cliffs, NJ : Prentice-Hall, 1979. (Русский перевод: Гейн К., Сарсон Т. Системный структурный анализ: средства и методы. – М. : Эйтекс, 1992.)
126. Garlan D. and Shaw M. An introduction to software architecture // *Advances in Software Engineering and Knowledge Engineering.* – 1993. – **1**. – P. 1–27.
127. Garlan D., Allen R. *et al.* Architectural mismatch: why reuse is so hard // *IEEE Software.* – 1995. – **12**(6). – P. 17–43.

128. Garlan D., Kaiser G. E. *et al.* Using tool abstraction to compose systems // *IEEE Computer.* – 1992. – **25**(6). – P. 30–38.
129. Gilb T. and Graham D. *Software Inspection.* – Wokingham Addison-Wesley, 1993.
130. Goldberg A. and Robson D. *Smalltalk-80. The Language and its Implementation.* – Reading, MA : Addison-Wesley, 1983.
131. Gollmann D. *Computer Security.* – Chichester : John Wiley and Sons, 1999.
132. Gomaa, H. *Software Design Methods for Concurrent and Real-time Systems.* – Reading, MA : Addison-Wesley, 1993.
133. Gordon V. S. and Bieman J. M. Rapid prototyping: lessons learned // *IEEE Software.* – 1995. – **12**(1). – P. 85–180.
134. Gotterbarn D., Miller K. *et al.* Software engineering code of ethics is approved // *Comm. ACM.* – 1999. – **42**(10). – P. 102–109.
135. Grady R. B. Practical results from measuring software quality // *Ibid.* – 1993. – **36**(11). – P. 62–70.
136. Grady R. B. and Van Slack. T. Key lessons in achieving widespread inspection use // *IEEE Software.* – 1994. – **11**(4). – P. 46–103.
137. Graham I. *Object-Oriented Methods, 2nd ed.* – Wokingham : Addison-Wesley, 1994.
138. Greenbaum J. and Kyng M. *Design at Work: Cooperative Design of Computer Systems.* – Hillsdale, NJ : Lawrence Erlbaum Associates, 1991.
139. Griss M. L. and Wosser M. Making reuse work at Hewlett-Packard // *IEEE Software.* – 1995. – **12**(1). – P. 105–112.
140. Grudin J. The case against user interface consistency // *Comm. ACM.* – 1989. – **32**(10). – P. 1164–1237.
141. Guimaraes T. Managing application program maintenance expenditures // *Ibid.* – 1983. – **26**(10). – P. 739–785.
142. Gunning R. *Techniques of Clear Writing.* – New York : McGraw–Hill, 1962.
143. Guttag J. Abstract data types and the development of data structures // *Comm. ACM.* – 1977. – **20**(6). – P. 396–801.
144. Guttag J. V., Horning J. J. et al. The Larch family of specification languages // *IEEE Software.* – 1985. – **2**(5). – P. 24–70.
145. Guttag J., Horning J. et al. *Larch: Languages and Tools for Formal Specification.* – Heidelberg : Springer-Verlag, 1993.
146. Haase V., Messnarz R. *et al.* Bootstrap: fine tuning process assessment // *IEEE Software.* – 1994. – **11**(4). – P. 25–60.
147. Hall A. Seven myths of formal methods // *IEEE Software.* – 1990. – **7** (5). – P. 11–31.
148. Hall A. Using formal methods to develop an ATC information system // *Ibid.* – 1996. – **13**(2). – P. 66–142.
149. Hall E. *Managing Risk: Methods for Software Systems Development.* – Reading, MA : Addison-Wesley Longman, 1998.
150. Hall T. and Fenton N. Implementing effective software metrics programs // *IEEE Software.* – 1997. – **14**(2). – P. 55–119.
151. Halstead M. H. *Elements of Software Science.* – Amsterdam : North-Holland, 1977.
152. Hamlet D. Are we testing for true reliability? // *IEEE Software.* – 1992. – **9**(4). – P. 21–58.
153. Hammer M. Reengineering work: don't automate, obliterate // *Harvard Business Review.* – July–August 1990. – P. 104–216.
154. Hammer M. and McLeod D. Database descriptions with SDM: a semantic database model // *ACM Trans. on Database Systems.* – 1981. – **6**(3). – P. 351–437.
155. Harel D. Statecharts: a visual formalism for complex systems // *Sci. Comput. Programming.* – 1987. – **8**(3). – P. 231–305.
156. Harel D. On visual formalisms // *Comm. ACM.* – 1988. – **31**(5). – P. 514–544.

157. Harker S. D. P., Easton K. D. *et al.* The change and evolution of requirements as a challenge to the practice of software engineering. – In : *Proc. RE'93.* – San Diego, USA, 1993.
158. Hayes I. (ed.). *Specification Case Studies.* – London : Prentice-Hall, 1987.
159. Heath C. and Luff P. Collaborative activity and technological design: task coordination in the London Underground control room. – In : *Proc. ECSCW'91.* – Amsterdam, Kluwer, 1991.
160. Heninger K. L. Specifying software requirements for complex systems. New techniques and their applications // *IEEE Trans. on Software Engineering.* – 1980. – **SE-6**(1). – P. 2–15.
161. Hihn J. and Habib-agahi H. Cost estimation of software intensive projects: a survey of current practices. – In : *Proc. 13th Int. Conf. on Software Engineering.* – Austin TX, ACM Press, 1991.
162. Hoare C. A. R. Monitors: an operating system structuring concept // *Comm. ACM.* – 1974. – **21**(8). – P. 666–743.
163. Hoare C. A. R. *Communicating Sequential Processes.* – London : Prentice-Hall, 1985.
164. Huff C. and Martin C. D. Computing consequences: a framework for teaching ethical computing // *Comm. ACM.* – 1995. – **38**(12). – P. 75–159.
165. Huff C. C. Elements of a realistic CASE tool adoption budget // *Ibid.* – 1992. – **35** (4). – P. 45–99.
166. Huff K. E. Software process modeling. – In *Trends in Software: Software Process* (A. Fuggetta and A. Wolf, eds). – New York : John Wiley and Sons, 1996.
167. Hughes J. A., O'Brien J. *et al.* Designing with ethnography: a presentation framework for design. – In : *Proc. DIS'97.* – Amsterdam : ACM Press, 1997.
168. Hughes J. A., Randall D. *et al.* Faltering from' ethnography to design. – In : *Proc. CSCW'92.* – Toronto, Canada, 1992.
169. Hughes J., Rodden T. *et al.* Moving out from the control room: ethnography in system design. – In : *Proc. CSCW'94.* – Greensborough, North Carolina, 1994.
170. Hull R. and King R. Semantic database modeling: survey, applications and research issues // *ACM Computing Surveys.* – 1987. – **19**(3). – P. 201–261.
171. Humphrey W. *Managing the Software Process.* – Reading, MA : Addison-Wesley, 1989.
172. Humphrey W. and Curtis B. Comment on 'A Critical Look' // *IEEE Software.* – 1991. – 8(4). – P. 42–89.
173. Humphrey W. S. Characterizing the software process // *Ibid.* – 1988. – 5(2). – P. 73–152.
174. Humphrey W. S. *A Discipline for Software Engineering.* – Reading, MA : Addison-Wesley, 1995.
175. IEC. *Draft Standard IEC 6150: Functional safety of electrical/electronic/programmable electronic safety-related systems.* International Electrotechnical Commission. – Geneva, 1998.
176. IEEE recommended practice for software requirements specifications. – In : *Software Requirements Engineering* (R. H. Thayer and M. Dorfman, eds). – Los Alamitos, CA : IEEE Computer Society Press, 1993.
177. IEEE. *Software Engineering Standards Collection.* – Los Alamitos CA : IEEE Press, 1994.
178. Ince D. *ISO 9001 and Software Quality Assurance.* – London : McGraw-Hill, 1994.
179. Ince D. C. and Hekmatpour S. Software prototyping – progress and prospects // *Information and Software Technology.* – 1987. – **29**(1). – P. 8–22.
180. ISO/IEC. *Information technology – Guidelines for the management of IT Security.* – ISO/IEC, 1998.
181. Jackson M. A. *System Development.* – London : Prentice-Hall, 1983.
182. Jackson M. A. *Requirements and Specifications.* – Wokingham : Addison-Wesley, 1995.
183. Jacky J. Specifying a safety-critical control system // *IEEE Trans. on Software Engineering.* – 1995. – **21**(2). – P. 99–205.
184. Jacky J. *The Way of Z: Practical Programming with Formal Methods.* – Cambridge : Cambridge University Press, 1997.
185. Jacky J., Unger J. *et al.* Experience with Z: developing a control program for a radiation therapy machine. – In : *Proc. ZUM'97.* – Reading : Springer, 1997.

186. Jacobson I., Christerson M. *et al. Object-Oriented Software Engineering.* – Wokingham : Addison-Wesley, 1993.
187. Jacobson I., Griss M. *et al. Software Reuse.* – Reading, MA : Addison-Wesley, 1997.
188. Jahanian F. and Mok A. K. Safety analysis of timing properties in real-time systems // *IEEE Trans.on Software Engineering.* – 1986. – **SE-12**(9). – P. 890–1794.
189. Janis I. L. *Victims of Groupthink. A Psychological Study of Foreign Policy Decisions and Fiascos.* – Boston : Houghton Mifflin, 1972.
190. Jelinski Z. and Moranda P. B. Software reliability research. – In : *Statistical Computer Performance Evaluation* (W. Frieberger, ed.). – New York : Academic Press, 1972. – P. 465–549.
191. Johnson P. L. *ISO 9000: Meeting the New International Standards.* – New York : McGraw-Hill, 1993.
192. Jones C. B. *Software Development – A Rigorous Approach.* – London : Prentice-Hall, 1980.
193. Jones C. B. *Systematic Software Development Using VDM.* – London : Prentice-Hall, 1986.
194. Jorgensen P. C. and Erickson C. Object-oriented integration testing // *Comm. ACM.* – 1994. – **37**(9). – P. 30–38.
195. Kafura D. and Reddy G. R. The use of software complexity metrics in software maintenance // *IEEE Trans. on Software Engineering.* – 1987. – **SE-13**(3). – P. 335–378.
196. Kemerer C. F. Reliability of function points measurement: a field experiment // *Comm. ACM.* – 1993. – **36**(2). – P. 85–182.
197. Kit E. *Software Testing in the Real World: Improving the Process.* – Reading, MA : Addison-Wesley, 1995.
198. Kitchenham B. Measuring software development. – In : *Software Reliability Handbook* (P. Rook, ed.). – Amsterdam : Elsevier, 1990. – P. 303–334.
199. Kitchenham B. Software development cost models. – In : *Software Reliability Handbook* (P. Rook, ed.). – Amsterdam : Elsevier, 1990. – P. 487–1004.
200. Knight J. C, and Leveson N. G. An experimental evaluation of the assumption of independence in multi-version programming // *IEEE Trans. on Software Engineering.* – 1986. – **SE-12**(1). – P. 96–205.
201. Knuth D. E. *The Art of Computer Programming: Fundamental Algorithms.* – Reading, MA : Addison-Wesley, 1971. (Русский перевод: Кнут Д. Искусство программирования. Т. 1. Основные алгоритмы. – М. : Изд. дом "Вильямс", 2000.)
202. Kotonya G. and Sommerville I. Viewpoints for requirements definition // *BCS/IEE Software Engineering J.* – 1992. – 7(6). – P. 375–462.
203. Kotonya G. and Sommerville I. Requirements engineering with viewpoints // *Ibid.* – 1996. – **11**(1). – P. 5–23.
204. Kotonya G. and Sommerville I. *Requirements Engineering: Processes and Techniques.* – Chichester, UK : John Wiley and Sons, 1998.
205. Kramer J. and Magee J. Dynamic configuration for distributed systems // *IEEE Trans. on Software Engineering.* – 1985. – **SE-11**(4). – P. 424–459.
206. Kuvaja P., Simila J. *et al. Software Process Assessment and Improvement: The BOOTSTRAP Approach.* – Oxford : Blackwell Publishers, 1994.
207. Kyng M. Designing for a dollar a day. – In : *Proc. CSCW'88.* – Portland OR : ACM Press, 1988.
208. Lamping J., Rao R. *et al.* A focus + context technique based on hyperbolic geometry for visualising large hierarchies. – In : *Proc. CHI'95.* – ACM Press, 1995.
209. Laprie J.-C., Arlat J. *et al.* Architectural issues in software fault tolerance. – In : *Software Fault Tolerance* (M. R. Lyu, ed.). – Chichester : John Wiley and Sons, 1995. – P. 47–127.
210. Laudon K. Ethical concepts and information technology // *Comm. ACM.* – 1995. – **38**(12). – P. 33–42.
211. Leblang D. The CM challenge: configuration management that works. – In : *Configuration Management* (W. Tichy, ed.). – Chichester : John Wiley and Sons, 1994. – P. 1–39.

212. Leblang D. B. and Chase R. P. Parallel software confiration management in a network environment // *IEEE Software.* – 1987. – 4(6). – P. 28–63.
213. Lehman M. M. and Belady L. *Program Evolution: Processes of Software Change.* – London : Academic Press, 1985.
214. Leveson N. G. Software safety. – In : *Resilient Computing Systems* (T. Anderson, ed.). – London : Collins, 1985. – P. 123–166.
215. Leveson N. G. Software safety: why, what and how // *ACM Computing Surveys.* – 1986. – 18(2). – P. 125–188.
216. Leveson N. G. *Safeware: System Safety and Computers.* – Reading, MA : Addison-Wesley, 1995.
217. Leveson N. G. and Harvey P. R. Analysing software safety // *IEEE Trans. on Software Engineering.* – 1983. – **SE-9**(5). – P. 569–648.
218. Lientz B. P. and Swanson E. B. *Software Maintenance Management.* – Reading, MA : Addison-Wesley, 1980.
219. Linger R. C. Cleanroom process model // *IEEE Software.* – 1994. – **11**(2). – P. 50–58.
220. Liskov B. and Guttag J. *Abstraction and Specification in Program Development.* – Cambridge, MA : MIT Press, 1986.
221. Littlewood B. Software reliability growth models. – In : *Software Reliability Handbook* (P. Rook, ed.). – Amsterdam : Elsevier, 1990. – P. 401–813.
222. Littlewood B. and Verrall J. L. A Bayesian reliability growth model for computer software // *Applied Statistics.* – 1973. – **22**. – P. 332–378.
223. Londeix B. *Cost Estimation for Software Development.* – Wokingham : Addison-Wesley, 1987.
224. Luqi Computer-aided prototyping for a command and control system using CAPS // *IEEE Software.* – 1992. – 9(1). – P. 56–113.
225. Lutz M. *Programming Python.* – Sebastopol, CA : O'Reilly and Associates, 1996.
226. Lutz R. R. Analysing software requirements errors in safety-critical embedded systems. – In : *Proc. RE'93.* – San Diego, CA : IEEE, 1993.
227. Lyu M. (ed.) *Software Reliability Engineering.* – New York : McGraw-Hill, 1996.
228. MacDonell S. G. Comparative review of functional complexity assessment methods for effort estimation // *BCS/IEE Software Engineering J.* – 1994. – 9(3). – P. 107–124.
229. Marshall J. E. and Heslin R. Boys and girls together: sexual composition and the effect of density on group size and cohesiveness // *J. of Personality and Social Psychology.* – 1975. – 35(5). – P. 952–1013.
230. Maslow A. A. *Motivation and Personality.* – New York : Harper and Row, 1954.
231. Matsumoto Y. Some experience in promoting reusable software: presentation in higher abstract levels // *IEEE. Trans. on Software Engineering.* – 1984. – **SE-10**(5). – P. 502–514.
232. McCabe T. J. A complexity measure // *Ibid.* – 1976. – **SE-2**(4). – P. 308–328.
233. McCue G. M. IBM's Santa Teresa Laboratory: architectural design for program development // *IBM Systems J.* – 1978. – **17**(1). – P. 4–29.
234. McGuffin R. W., Elliston A. E. *et al.* CADES – Software engineering in practice. – In : *Proc. 4th Int. Conf. on Software Engineering.* – Munich, Germany, 1979.
235. McKee J. R. Maintenance as a function of design. – In : *AFIPS National Computer Conf.* – Las Vegas, 1984.
236. Meyer B. On to components // *IEEE Computer.* – 1999. – **32**(1). – P. 139–179.
237. Miller G. A. The magical number 7 plus or minus two: some limits on our capacity for processing information // *Psychological Review.* – 1957. – **63**. – P. 81–178.
238. Millington D. and Stapleton J. Special report: developing a RAD Standard // *IEEE Software.* – 1995. – **12**(5). – P. 54–60.
239. Mills H. D., Dyer M. *et al.* Cleanroom software engineering // *Ibid.* – 1987. – 4(5). – P. 19–44.

240. Mills H. D., O'Neill D. *et al.* The management of software engineering // *IBM Sys. J.* – 1980. – **24**(2). – P. 414–493.
241. MOD. *The Procurement of Safety Critical Software (Revised ed.).* – UK Ministry of Defence, Procurement Executive, 1995.
242. Mullery G. CORE – a method for controlled requirements specification. – In : *Proc. 4th Int. Conf. on Software Engineering.* – Munich : IEEE Press, 1979.
243. Mumford E. User participation in a changing environment – why we need it. – In : *Participation in Systems Development* (K. Knight, ed.). – London : Kogan Page, 1989.
244. Munch B. P., Larsen J.-L. *et al.* Uniform versioning: the change-oriented model. – In : *Proc. 4th Workshop on Software Configuration Management.* – Baltimore, MD, 1993.
245. Murphy G. C., Townsend P. *et al.* Experiences with cluster and class testing // *Comm. ACM.* – 1994. – **37**(9). – P. 39–86.
246. Musa J. D. Operational profiles in software reliability engineering // *IEEE Software.* – 1993. – **10**(2). – P. 14–46.
247. Musa J. D. *Software Reliability Engineering: More Reliable Software, Faster Development and Testing.* – New York : McGraw-Hill, 1998.
248. Musa J. D., Iannino A. *et al. Software Reliability: Measurement, Prediction, Application.* – New York : McGraw-Hill, 1987.
249. Musciano C. and Kennedy B. *HTML The Definitive Guide.* – Sebastopol, CA : O'Reilly and Associates Inc., 1998.
250. Myers G. J. *Reliable Software through Composite Design.* – New York : Petrocelli, 1975. (Русский перевод: Майерс Г. Надежность программного обеспечения. – М. : Мир, 1980.)
251. Myers W. Allow plenty of time for large-scale software // *IEEE Software.* – 1989. – **6**(4). – P. 92–101.
252. Nakajo T. and Kume H. A case history analysis of software error-cause relationships // *IEEE Trans. on Software Engineering.* – 1991. – **18**(8). – P. 830–838.
253. Neil M., Ostrolenk G. et al. Lessons from using Z to specify a software tool // *IEEE Trans. on Software Engineering.* – 1998. – **24**(1). – P. 15–38.
254. Neilsen J. *Usability Engineering.* – New York : Academic Press, 1993.
255. Nii H. P. Blackboard systems, Parts 1 and 2 // *AI Magazine.* – 1986. – **7**(3 and 4). – P. 38–92 and 62–71.
256. Nilsen K. Adding real-time capabilities to Java // *Comm. ACM.* – 1998. – **41**(6). – P. 49–105.
257. Ning J. Q., Engberts A. *et al.* Automated support for legacy code understanding // *IEEE Software.* – 1994. – **37**(5). – P. 50–57.
258. Norman D. A. and Draper S. W. *User-centered System Design.* – Hillsdale, NJ : Lawrence Erlbaum, 1986.
259. Nosek J. T. and Palvia P. Software maintenance management: changes in the last decade // *Software Maintenance: Research. and Practice.* – 1990. – **2**(3). – P. 157–231.
260. Nuseibeh B., Kramer J. *et al.* A framework for expressing the relationships between multiple views in requirements specifications // *IEEE Trans. on Software Engineering.* – 1994. – **20**(10). – P. 760–833.
261. O'Connor J., Mansour C. *et al.* Reuse in command and control systems // *Ibid.* – 1994. – **11**(4). – P. 70–79.
262. Offen R. J. and Jeffrey R. Establishing software measurement programs // *IEEE Software.* – 1997. – **14**(2). – P. 45–99.
263. Oman P. W. and Cook C. R. The book paradigm for improved maintenance // *Ibid.* – 1990. – **7**(1). – P. 39–84.
264. Orfali R. and Harkey D. *Client/server Programming with Java and CORBA.* – New York : John Wiley and Sons, 1998.

265. Oskarsson O. and Glass R. L. *An ISO 9000 Approach to Building Quality Software.* – Englewood Cliffs, NJ : Prentice-Hall, 1995.
266. Ould M. *Managing Software Quality and Business Risk.* – Chichester : John Wiley and Sons, 1999.
267. Ousterhout J. *Tcl and the Tk toolkit.* – Reading, MA : Addison-Wesley, 1994.
268. Ousterhout J. Scripting: higher-level programming for the 21st century // *IEEE Computer.* – 1998. – **31**(3). – P. 23–53.
269. Park R. *Software Size Measurement: A Framework for Counting Source Statements.* – Software Engineering Institute, 1992.
270. Parnas D. L., van Schouwen J. *et al.* Evaluation of safety-critical software // *Comm. ACM.* – 1990. – **33**(6). – P. 636–687.
271. Paulk M. How ISO 9000 compares with the CMM // *IEEE Software.* – 1995. – **12**(1). – P. 74–158.
272. Paulk M. C. and Konrad M. An overview of ISO's SPICE Project // *IEEE Computer.* – 1994. – **27**(4). – P. 68–138.
273. Paulk M. C., Curtis B. *et al.* Capability Maturity Model, Version 1.1 // *IEEE Software.* – 1993. – **10**(4). – P. 18–45.
274. Peach R. W. *The ISO 9000 Handbook (3rd edn).* – New York : Irwin Professional Publishers, 1996.
275. Perrow C. *Normal Accidents: Living with High-Risk Technology.* – New York : Basic Books, 1984.
276. Perry W. *Effective Methods for Software Testing.* – New York : John Wiley and Sons, 1995.
277. Peterson J. L. *Petri Net Theory and the Modeling of Systems.* – New York : McGraw-Hill, 1981. (Русский перевод: Питерсон Д. Теория сетей Петри и моделирование систем. – М. : Мир, 1984.)
278. Pfaff G. and ten Hagen P. J. W. *Seeheim Workshop on User Interface Management Systems.* – Heidelberg : Springer-Verlag, 1985.
279. Pfleeger C. P. *Security in Computing, 2nd edn.* – Englewood Cliffs : Prentice-Hall, 1997.
280. Pope A. *CORBA.* – Harlow, UK : Addison-Wesley Longman, 1998.
281. Preece J., Rogers Y. *et al. Human-Computer Interaction.* – Wokingham : Addison-Wesley, 1994.
282. Pressman R. S. *Software Engineering: A Practitioner's Approach.* – New York : McGraw Hill, 1997.
283. Prieto-Dfaz R. and Arango G. *Domain Analysis and Software System Modeling.* – Los Alamitos CA : IEEE Press, 1991.
284. Prowell S. J., Trammell C. J. *et al. Cleanroom Software Engineering: Technology and Process.* – Reading, MA : Addison-Wesley Longman, 1999.
285. Pulford K., Kuntzmann-Combelles A. *et al. A Quantitative Approach to Software Management.* – Wokingham : Addison-Wesley, 1996.
286. Putnam L. H. A general empirical solution to the macro software sizing and estimating problem // *IEEE Trans. on Software Engineering.* – 1978. – **SE-4**(3). – P. 345–406.
287. Pycock J. and Bowers J. Getting others to get it right: an ethnography of design work in the fashion industry. – In : *Proc. CSCW'96.* – Boston : ACM Press, 1996.
288. Randell B. System structure for software fault tolerance // *IEEE Trans. on Software Engineering.* – 1975. – **SE-1**(2). – P. 220–252.
289. Randell B. and Xu J. The evolution of the recovery block concept. – In : *Software Fault Tolerance* (M. R. Lyu, ed.). – Chichester : John Wiley and Sons, 1995. – P. 1–23.
290. Redmill F. IEC 61508: Principles and use in the management of safety // *IEEE Computing and Control Engineering J.* – 1998. – **9**(10). – P. 205–218.
291. Reiss S., P. Connecting tools using message passing in the field environment // *IEEE Software.* – 1990. – **7**(4). – P. 57–123.
292. Rettig M. Practical programmer: prototyping for tiny fingers // *Comm. ACM.* – 1994. – **37**(4). – P. 21–28.
293. Rickets J. A., DelMonaco J. C. *et al.* Data reengineering for application systems. – In : *Software Reengineering* (R. S. Arnold, ed.). – Los Alamitos CA : IEEE Press, 1993. – P. 288–381.

294. Rittel H. and Webber M. Dilemmas in a general theory of planning // *Policy Sciences.* – 1973. – 4. – P. 155–224.
295. Robinson P. J. *Hierarchical Object-Oriented Design.* – Englewood Cliffs, NJ : Prentice-Hall, 1992.
296. Rochester J. B. and Douglass D. P. Re-engineering existing systems. – In : *Software Reengineering* (R. S. Arnold, ed.). – Los Alamitos, CA : IEEE Press, 1993. – P. 41–92.
297. Rochkind M. J. The Source Code Control System // *IEEE Trans. on Software Engineering.* – 1975. – **SE-1**(4). – P. 255–320.
298. Rockwell R. and Gera M. H. The Eureka Software Factory CoRe: a conceptual reference model for software factories. – In : *Proc. 6th Conf. on Software Engineering Environments.* – Reading, UK : IEEE Press, 1993.
299. Ross D. T. Structured analysis (SA). A language for communicating ideas // *IEEE Trans. on Software Engineering.* – 1977. – **SE-3**(1). – P. 16–50.
300. Royce W. W. Managing the development of large software systems: concepts and techniques. – In : *Proc. IEEE WESTCON.* – Los Angeles, CA, 1970.
301. Rubin K. and Goldberg A. Object behaviour analysis // *Comm. ACM.* – 1992. – **35**(9). – P. 48–110.
302. Rumbaugh J., Blaha M. *et al. Object-oriented Modeling and Design.* – Englewood Cliffs, NJ : Prentice-Hall, 1991.
303. Rumbaugh J., Jacobson I. *et al. The Unified Modeling Language Reference Manual.* – Reading, MA : Addison-Wesley, 1999.
304. Rumbaugh J., Jacobson I. *et al. The Unified Software Development Process.* – Reading, MA : Addison-Wesley, 1999.
305. Sackman H., Erikson W. J. *et al.* Exploratory experimentation studies comparing on-line and off line programming performance // *Comm. ACM.* – 1968. – **11**(1). – P. 3–14.
306. Schmidt D. C. Applying design patterns and frameworks to develop object-oriented communications software. – In : *Handbook of Programming Languag, Vol. 1* (P. Salus, ed.). – New York : Macmillan Computer Publishing, 1997.
307. Schneidewind N. F. and Keller T. W. Applying reliability models to the space shuttle // *IEEE Software.* – 1992. – **9**(4). – P. 28–61.
308. Schoman K. and Ross D. T. Structured analysis for requirements definition // *IEEE Trans. on Software Engineering.* – 1977. – **SE-3**(1). – P. 6–21.
309. Selby R. Empirically analyzing software reuse in a production environment. – In : *Software Reuse: Emerging Technology* (W. Tracz, ed.). – Los Alamitos, CA : IEEE Computer Society Press, 1988.
310. Selby R. W., Basili V. R. *et al.* Cleanroom software development: an empirical evaluation // *IEEE Trans. on Software Engineering.* – 1987. – **SE-13**(9). – P. 1027–1064.
311. Sessions R. *COM and DCOM: Microsoft's Vision for Distributed Objects.* – New York : John Wiley and Sons, 1997.
312. Sheldon F. T., Kavi K. M. *et al.* Reliability measurement: from theory to practice // *IEEE Software.* – 1992. – **9**(4). – P. 13–33.
313. Shlaer S. and Mellor S. *Object-Oriented Systems Analysis: Modeling the World in Data.* – Englewood Cliffs, NJ : Yourdon Press, 1988.
314. Shneiderman B. *Software Psychology.* – Cambridge, MA : Winthrop Publishers Inc., 1980.
315. Shneiderman B. *Designing the User Interface, 1st edn.* – Reading, MA : Addison-Wesley, 1992.
316. Shneiderman B. *Designing the User Interface, 3rd edn.* – Reading, MA : Addison-Wesley, 1998.
317. Siegel J. OMG overview: CORBA and the OMA in enterprise computing // *Comm. ACM.* – 1998. – **41**(10). – P. 37–80.
318. Silberschaltz A. and Galvin P. *Operating System Concepts, 5th edn.* – Reading, MA : Addison-Wesley, 1998.

319. Soloway E., Ehrlich K. *et al.* What do novices know about programming? – In : *Directions in Human-Computer Interaction* (A. Badre and B. Shneiderman, eds). – Norwood, NJ : Ablex Publishing Co, 1982.
320. Sommerville I. and Dean G. PCL: a language for modelling evolving system architectures // *BCS/IEE Software Engineering J.* – 1996. – **11**(2). – P. 111–132.
321. Sommerville I. and Sawyer P. *Requirements Engineering: A Good Practice Guide.* – Chichester : John Wiley and Sons, 1997.
322. Sommerville I., Rodden T. *et al.* Integrating ethnography into the requirements engineering process. – In : *Proc. RE'93.* – San Diego, CA : IEEE Computer Society Press, 1993.
323. Spafford E. The Internet worm: crisis and aftermath // *Comm ACM.* – 1989. – **32**(6). – P. 678–765.
324. Spivey J. M. Specifying a real-time kernel // *IEEE Software.* – 1990. – **7**(5). – P. 21–29.
325. Spivey J. M. *The Z Notation: A Reference Manual, 2nd edn.* – London : Prentice-Hall, 1992.
326. St Laurent S. and Cerami E. *Building XML Applications.* – New York : McGraw-Hill, 1999.
327. Stapleton J. *DSDM Dynamic Systems Development Method.* – Harlow, UK : Addison-Wesley Longman, 1997.
328. Suchman L. Office procedures as practical action // *ACM Trans. on Office Information Systems.* – 1983. – **1**(3). – P. 320–348.
329. Swartz A. J. Airport 95: automated baggage system? // *ACM Software Engineering Notes.* – 1996. – **21**(2). – P. 79–162.
330. Symons C. R. Function-point analysis: difficulties and improvements // *IEEE Trans. on Software Engineering.* – 1988. – **14**(1). – P. 2–13.
331. Szyperski C. *Component Software: Beyond Object-oriented Programming.* – Reading, MA : Addison-Wesley, 1998.
332. Tanenbaum A. S. and Woodhull A. S. *Operating Systems: Design and Implementation, 2nd edn.* – Englewood Cliffs, NJ : Prentice-Hall, 1997.
333. Teichrow D. and Hershey E. A. PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems // *IEEE Trans. on Software Engineering.* – 1977. – **SE-3**(1). – P. 41–49.
334. Thayer R. H. and Dorfman M. *Software Requirements Engineering, 2nd ed.* – Los Alamitos, CA : IEEE Computer Society Press, 1997.
335. Tichy W. RCS – a system for version control // *Software Practice and Experience.* – 1985. – **15**(7). – P. 637–691.
336. Ulrich W. M. The evolutionary growth of software reengineering and the decade ahead // *American Programmer.* – 1990. – **3**(10). – P. 14–34.
337. Wall L., Christiansen T. *et al. Programming Perl.* – Sebastopol, CA : O'Reilly and Associates, 1996.
338. Ward P. and Mellor S. *Structured Development for Real-time Systems.* – Englewood Cliffs, NJ : Prentice Hall, 1985.
339. Warren I. (ed.) *The Renaissance of Legacy Systems.* – London : Springer, 1998.
340. Weinberg G. *The Psychology of Computer Programming.* – New York : Van Nostrand, 1971.
341. White S., Alford M. *et al.* Systems engineering of computer-based systems // *IEEE Computer.* – 1993. – **26**(11). – P. 54–119.
342. Whitgift D. *Software Configuration Management: Methods and Tools.* – Chichester : John Wiley and Sons, 1991.
343. Wirfs-Brock R. J. and Johnson R. E. Surveying current research in object-oriented design // *Comm. ACM.* – 1990. – **33**(9). – P. 104–128.
344. Wirfs-Brock R., Wilkerson B. *et al. Designing Object-Oriented Software.* – Englewood Cliffs, NJ : Prentice-Hall, 1990.

345. Wirth N. *Systematic Programming: An Introduction.* – Englewood Cliffs, NJ : Prentice-Hall, 1976. (Русский перевод: Вирт Н. Системное программирование: Введение. – М. :, Мир, 1977.)
346. Wood J. and Silver D. *Joint Application Development (2nd ed.).* – New York : John Wiley and Sons, 1995.
347. Woodcock J. C. P. and Davies J. *Using Z: Specification, Proof, Refinement.* – London : Prentice-Hall, 1996.
348. Wordsworth J. *Software Engineering with B.* – Wokingham : Addison-Wesley, 1996.
349. Wordsworth J. B. The CICS application programming interface definition. – In : *Proc. Z User Workshop.* – Oxford, Berlin : Springer, 1991.
350. Zave P. A compositional approach to multiparadigm programming // *IEEE Software.* – 1989. – 6(5). – P. 15–42.
351. Zave P. and Schell W. Salient features of an executable specification language and its environment // *IEEE Trans. on Software Engineering.* –1986. – **SE-12**(2). – P. 312–337.
352. Zimmermann H. OSI Reference Model – the ISO model of architecture for open systems interconnection // *IEEE Transactions on Communications.* – 1980. – **COM-28**(4). – P. 425–457.

Литература, добавленная при переводе[1]

1*. Андон А.И., Яшунин А.Е., Резниченко А.А. Логические модели интеллектуальных информационных систем. – К. : Наук. думка, 1999.
2*. Андон Ф.И., Лаврищева Е.М. Методы инженерии распределенных компьютерных систем. – К. : Наук. думка, 1997.
3*. Бабенко Л.П., Лавріщева К.М. Основи программної інженерії. Навч. посіб. – К. : Т-во "Знання", 2001.
4*. Бен-Ари М. Языки программирования. Практический сравнительный анализ. – М. : Мир, 2000.
5*. Боггс Ч., Боггс М. UML Rational. – М. : ЛОРИ, 2000.
6*. Вендров А.М. CASE-технологии. Современные методы и средства проектирования информационных систем. – М. : Финансы и статистика, 1998.
7*. Вендров А.М. Проектирование программного обеспечения экономических информационных систем. – М. : Финансы и статистика, 2000.
8*. Гласс Г. Руководство по надежному программированию. – М. : Финансы и статистика, 1982.
9*. Калянов Г.Н. CASE-технологии. Консалтинг при автоматизации бизнес-процессов, 2-е изд. – М. : Горячая линия – Телеком, 2000.
10*. Калянов Г.Н. Методы и средства системного структурного анализа и проектирования. – М. : НИВЦ МГУ, 1995.
11*. Кантор М. Управление программными проектами. – М. : Издат. дом "Вильямс", 2002.
12*. Коллинз Г., Блей Д. Структурные методы разработки систем: от стратегического планирования до тестирования. – М. : Финансы и статистика, 1986.
13*. Коуд П., Норт Д., Мейфилд М. Объектные модели. Стратегии, шаблоны и приложения. – М. : ЛОРИ, 1999.
14*. Кратчен Ф. Введение в Rational Unified Process, 2-е изд. – М. : Издат. дом "Вильямс", 2002.
15*. Кулаков А.Ф. Оценка качества программ ЭВМ. – К. : Техніка, 1984.
16*. Лаврищева Е.М., Грищенко В.Н. Сборочное программирование. – К. : Наук. думка, 1991.
17*. Ларман К. Применение UML и шаблонов проектирования, 2-е изд. – М. : Издат. дом "Вильямс", 2001.

[1] *В этот список включены в основном изданные в последние годы книги. Список журнальных публикаций по данной теме слишком обширный, чтобы включить его в качестве дополнения к библиографии автора. Отметим, что не на все приведенные в списке книги редактором сделаны ссылки в тексте, поскольку многие из них дополняют и расширяют материал автора.* – Прим. ред.

18*. Лефингвел Д., Уидриг Д. Принципы работы с требованиями к программному обеспечению. Унифицированный подход. – М. : Издат. дом "Вильямс", 2002.
19*. Липаев В.В. Документирование и управление конфигурацией программных средств. Методы и стандарты. – М. : СИНТЕГ, 1998.
20*. Липаев В.В. Надежность программных средств. – М. : СИНТЕГ, 1998.
21*. Липаев В.В. Отладка сложных программ. – М. : Энергоатомиздат, 1993.
22*. Липаев В.В. Тестирование программ. – М. : Радио и связь, 1986.
23*. Липаев В.В. Управление разработкой программных комплексов. – М. : Финансы и статистика, 1993.
24*. Марка Д.А., Мак-Гоуэн К. Методология структурного анализа и проектирования. – М. : МетаТехнология, 1993.
25*. Мацяшек Л. А. Анализ требований и разработка информационных систем с использованием UML. – М. : Издат. дом "Вильямс", 2002.
26*. Мороз Г.Б., Лаврищева Е.М. Модели роста надежности программного обеспечения. – К., 1992. – (Препринт / АНУ; Ин-т кибернетики ; 92-38).
27*. Слама Д., Гарбис Дж., Перри Р. Корпоративные системы на основе CORBA. – М. : Издат. дом "Вильямс", 1999.
28*. Спирли Э. Кооперативные хранилища данных. Планирование, разработка, реализацня. Т. 1. – М. : Издат. дом "Вильямс", 2001.
29*. Тейер Т., Липов М., Нельсон Э. Надежность программного обеспечения. – М. : Мир, 1981.
30*. Фаулер М., Скотт К. UML в кратком изложении. Применение стандартного языка объектного моделирования. – М. : Мир, 1999.
31*. Фокс Д. Программное обеспечение и его разработка. – М. : Мир, 1985.
32*. Шаллоуей А., Тротт Дж. Р. Шаблоны проектирования. Новый подход к объектно-ориентированной разработке. – М. : Издат. дом "Вильямс", 2002.
33*. Шлеер С., Меллор С. Объектно-ориентированный анализ: моделирование мира в состояниях. – К. : Диалектика, 1993.
34*. Элиенс А. Принципы объектно-ориентированной разработки программ, 2-е изд. – М. : Издат. дом "Вильямс", 2002.
35*. Юдицкий С.А., Барон Ю.Л., Жукова Г.Н. Построение и анализ логического портрета сложных систем. – М. : ИПУ, 1997.

Предметный указатель

Р

С

Научно-популярное издание

Иан Соммервилл

Инженерия программного обеспечения, 6-е издание

Литературный редактор	*Т.П. Кайгородова*
Верстка	*О.В. Мишутина*
Художественный редактор	*Е.П. Дынник*
Корректоры	*Л.А. Гордиенко, Т.А. Корзун*

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д. 43, стр. 624.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 21.08.02. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 56,76. Уч.-изд. л. 44,00.
Тираж 3500 экз. Заказ № 1117.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

ИАН СОММЕРВИЛЛ

Инженерия программного обеспечения

6-е издание

Данная книга является прекрасным введением в инженерию программного обеспечения. Здесь дана широкая панорама тем инженерии ПО, охватывающих все этапы и технологии разработки программных систем. В семи частях книги представлен весь спектр процессов, ведущих к созданию программного обеспечения, — от начальной разработки системных требований через проектирование, непосредственное программирование и аттестацию до модернизации программных систем. Книга окажет неоценимую поддержку студентам и аспирантам, изучающим дисциплину "Инженерия программного обеспечения", а также будет полезна тем специалистам по программному обеспечению, которые хотят познакомиться с новыми технологиями разработки спецификации требований, с архитектурой распределенных систем или методами создания надежного ПО.

Шестое издание значительно изменено по сравнению с пятым, добавлены новые темы, часть материала, включенного в предыдущее издание, удалена. В результате объем книги уменьшился на 10% по сравнению с пятым изданием.

Web-страница данной книги (http://*www.softwareengin.com*) содержит ссылки на материал, который используется в книге и будет полезен как преподавателям, так и изучающим инженерию программного обеспечения самостоятельно. Здесь можно найти руководство для преподавателей, включающее советы по использованию данной книги, рекомендации по аудиторной работе, задания для самостоятельного изучения, решения некоторых примеров из книги; иллюстративный материал для каждой главы; исходные коды программ для всех основных примеров из книги; дополнительный материал по CASE-средствам и формальным спецификациям.

Иан Соммервилл (Ian Sommerville), профессор инженерии программного обеспечения университета Ланкастера, Англия. Он имеет более чем 20-летний опыт преподавания и научной работы в области инженерии ПО. Круг его сегодняшних научных интересов включает инженерию компьютерных систем, спецификацию требований, вопросы надежности и эволюции программных систем.

Отличия от пятого издания:

- Добавлены новые главы, описывающие процессы разработки программного обеспечения, архитектуру распределенных систем, проблемы надежности и наследования ПО.
- В данном издании примеры программ написаны на языке Java, а модели объектов — на языке моделирования UML.
- Во все главы внесены изменения, а некоторые существенно переработаны. Повторное использование ПО в настоящем издании рассматривается как тема разработки ПО с повторным использованием программных компонентов совместно с материалом о паттернах и темой покомпонентной разработки ПО. Рассмотрены объектно-ориентированное проектирование и использование языка моделирование UML. Главы, посвященные системным требованиям, охватывают как описание требований, так и процесс их разработки. Материал об оценивании стоимости ПО переписан и теперь включает описание модели COCOMO 2.
- Материал по критическим системам реструктуризирован, темы надежности, работоспособности, безопасности и защищенности ПО раскрыты в главах, посвященных спецификированию, разработке и аттестации критических систем.
- Теме формальной спецификации ПО теперь посвящена только одна глава. Материал по CASE-технологиям сокращен и распределен по нескольким главам. Материал по функциональному проектированию ПО включен в новую главу по наследуемым системам.

ISBN 5-8459-0330-0

02100

РУБ 259.00

9 785845 903303

www.pearsoneduc.com

www.williamspublishing.com