Министерство образования и науки Российской Федерации

Государственное образовательное учреждение
высшего профессионального образования
«Тамбовский государственный технический университет»

И.В. МИЛОВАНОВ, В.И. ЛОСКУТОВ

ОСНОВЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Утверждено Учёным советом университета
в качестве учебного пособия
для студентов 2 – 4 курсов дневного отделения
специальностей 231000, 230100, 010400

Тамбов
Издательство ГОУ ВПО ТГТУ
2011

УДК 004.415.2
ББК 32.973.26.32.973.26-018.2
М605

Р е ц е н з е н т ы:

Доктор технических наук, профессор,
проректор по информатизации ГОУ ВПО ТГТУ
В.Е. Подольский

Доктор технических наук, профессор,
заведующий кафедрой «Компьютерное и математическое моделирование»
ГОУ ВПО ТГУ им. Г.Р. Державина
А.А. Арзамасцев

Милованов, И.В.

М605 Основы разработки программного обеспечения вычислительных систем : учебное пособие / И.В. Милованов, В.И. Лоскутов. – Тамбов : Изд-во ГОУ ВПО ТГТУ, 2011. – 88 с. – 100 экз.
ISBN 978-5-8265-0990-6.

Изложены основы проектирования программных систем, используемых в инженерном цикле разработки сложных программных продуктов. Представлены классические основы проектирования программных систем, показан объектно-ориентированный подход к разработке программного обеспечения, а также язык визуального моделирования объектных систем, рассмотрены различные подходы к конструированию программных комплексов.

Предназначено для студентов 2 – 4 курсов дневного отделения специальностей 231000 «Программная инженерия», 230100 «Информатика и вычислительная техника», 010400 «Прикладная математика и информатика».

УДК 004.415.2
ББК 32.973.26.32.973.26-018.2

ISBN 978-5-8265-0990-6

ВВЕДЕНИЕ

В настоящее время вычислительные системы находят всё более и более широкое применение. При этом, программное обеспечение (ПО) является неотъемлемой частью таких систем. Программные системы весьма сложны, например, операционные системы и системы автоматизированного проектирования, другие программы, как системы домашней бухгалтерии, наоборот ясны и понятны широкому кругу пользователей.

При всём многообразии программ и программных комплексов у них есть одна общая черта – технологии разработки. В 1969 г. фирма IBM разделила аппаратную и программную части вычислительной системы, положив начало индустрии программного обеспечения, а также подходам, методам, средствам и технологиям разработки программ.

Учебное пособие посвящено основам проектирования программных систем, объектно-ориентированному подходу к реализации систем и инструментарию построения объектно-ориентированных моделей.

В первом разделе рассматривается содержание этапа проектирования и его место в жизненном цикле конструирования программных систем. Даётся обзор архитектурных моделей ПО, обсуждаются классические проектные характеристики: модульность, информационная закрытость, сложность, связность, сцепление и метрики для их оценки.

Второй раздел вводит в круг вопросов объектно-ориентированного представления программных систем. В этой главе рассматриваются: абстрагирование понятий проблемной области, приводящее к формированию классов; инкапсуляция объектов, обеспечивающая скрытность их характеристик; модульность как средство упаковки набора классов; особенности построения иерархической структуры объектно-ориентированных систем. Последовательно обсуждаются объекты и классы как основные строительные элементы объектно-ориентированного ПО. Значительное внимание уделяется описанию отношений между объектами и классами.

Третий раздел посвящён определению базовых понятий языка визуального моделирования UML.

Учебное пособие предназначено для студентов и бакалавров направлений «Программная инженерия», «Информатика и вычислительная техника», «Прикладная математика и информатика» и других направлений, изучающих технологии разработки программных систем.

1. ОСНОВЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ

ОСОБЕННОСТИ ПРОЦЕССА СИНТЕЗА ПРОГРАММНЫХ СИСТЕМ

Известно, что технологический цикл конструирования программной системы (ПС) включает три процесса – анализ, синтез и сопровождение.

В ходе анализа определяется ответ на вопрос: «Что должна делать будущая система?». Именно на этой стадии закладывается фундамент успеха всего проекта. Известно множество неудачных реализаций из-за неполноты и неточностей в определении требований к системе.

В процессе синтеза формируется ответ на вопрос: «Каким образом система будет реализовывать предъявляемые к ней требования?». Выделяют три этапа синтеза: проектирование ПС, кодирование ПС, тестирование ПС (рис. 1.1).

Рассмотрим информационные потоки процесса синтеза.

Этапы проектирования опираются на требования к ПС, представленные информационной, функциональной и поведенческой моделями анализа. Иными словами, модели анализа поставляют этапу проектирования исходные сведения для работы. Информационная модель описывает инфор-

Рис. 1.1. Информационные потоки процесса синтеза ПС

мацию, которую, по мнению заказчика, должна обрабатывать ПС. Функциональная модель определяет перечень функций обработки. Поведенческая модель фиксирует желаемую динамику системы (режимы её работы). На выходе этапа проектирования – разработка данных, разработка архитектуры и процедурная разработка ПС.

Разработка данных – это результат преобразования информационной модели анализа в структуры данных, которые потребуются для реализации программной системы.

Разработка архитектуры выделяет основные структурные компоненты и фиксирует связи между ними.

Процедурная разработка описывает последовательность действий в структурных компонентах, т.е. определяет их содержание.

Далее создаются тексты программных модулей, проводится тестирование для объединения и проверки ПС. На проектирование, кодирование и тестирование приходится более 75% стоимости конструирования ПС. Принятые здесь решения оказывают решающее воздействие на успех реализации ПС и лёгкость, с которой ПС будет сопровождаться.

Следует отметить, что решения, принимаемые в ходе проектирования, делают его стержневым этапом процесса синтеза. Важность проектирования можно определить одним словом – качество. Проектирование – этап, на котором «выращивается» качество разработки ПС. Справедлива следующая аксиома разработки: может быть плохая ПС при хорошем проектировании, но не может быть хорошей ПС при плохом проектировании. Проектирование обеспечивает нас такими представлениями ПС, качество которых можно оценить. Проектирование – единственный путь, обеспечивающий правильную трансляцию требований заказчика в конечный программный продукт.

ОСОБЕННОСТИ ЭТАПА ПРОЕКТИРОВАНИЯ

Проектирование – итерационный процесс, при помощи которого требования к ПС транслируются в инженерные представления ПС. Вначале эти представления дают только концептуальную информацию (на высоком уровне абстракции), последующие уточнения приводят к формам, которые близки к текстам на языках программирования.

Обычно в проектировании выделяют две ступени: предварительное проектирование и детальное проектирование. Предварительное проектирование формирует абстракции архитектурного уровня, детальное проектирование уточняет эти абстракции, добавляет подробности алгоритмического уровня. Кроме того, во многих случаях выделяют интерфейсное проектирование, цель которого – сформировать графический интерфейс пользователя (GUI). Схема информационных связей процесса проектирования приведена на рис. 1.2.

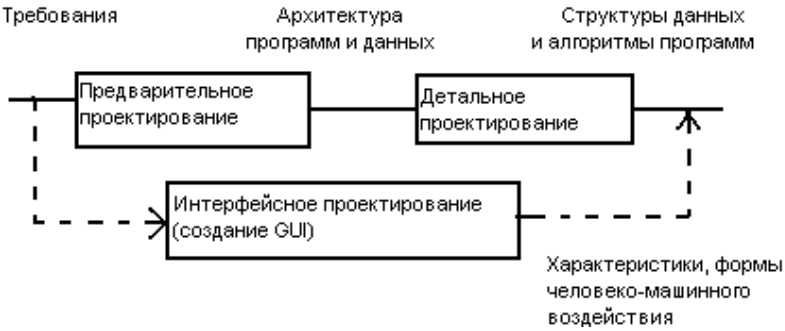

Рис. 1.2. Информационные связи процесса проектирования

Предварительное проектирование обеспечивает:

- идентификацию подсистем;
- определение основных принципов управления подсистемами, взаимодействия подсистем.

Предварительное проектирование включает три типа деятельности:

1. *Структурирование системы*. Система структурируется на несколько подсистем, где под подсистемой понимается независимый программный компонент. Определяются взаимодействия подсистем.
2. *Моделирование управления*. Определяется модель связей управления между частями системы.
3. *Декомпозиция подсистем на модули*. Каждая подсистема разбивается на модули. Определяются типы модулей и межмодульные соединения.

Рассмотрим вопросы структурирования, моделирования и декомпозиции более подробно.

СТРУКТУРИРОВАНИЕ СИСТЕМЫ

Известны четыре модели системного структурирования:

- модель хранилища данных;
- модель клиент-сервер;
- трёхуровневая модель;
- модель абстрактной машины.

В *модели хранилища данных* (рис. 1.3) подсистемы разделяют данные, находящиеся в общей памяти. Как правило, данные образуют базу данных (БД). Предусматривается система управления этой базой.

Модель клиент-сервер используется для распределённых систем, где данные распределены по серверам (рис. 1.4). Для передачи данных применяют сетевой протокол, например TCP/IP.

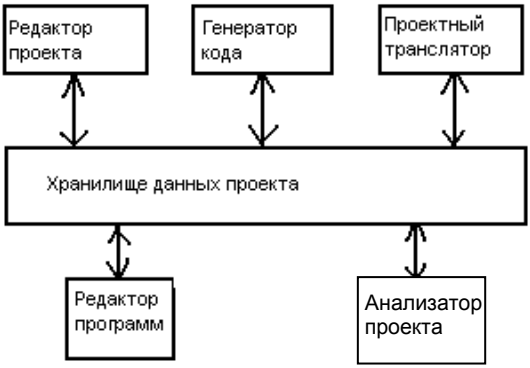

Рис. 1.3. Модель хранилища данных

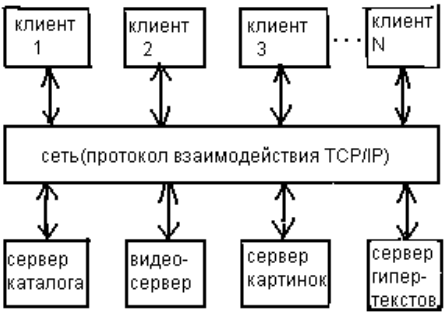

Рис. 1.4. Модель клиент-сервер

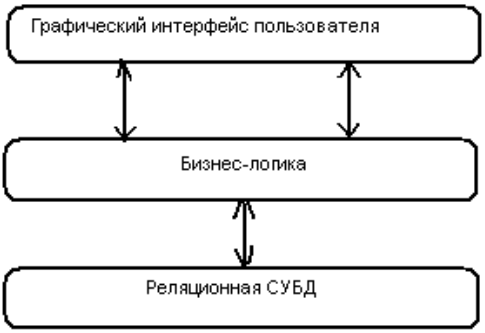

Рис. 1.5. Трёхуровневая модель

Трёхуровневая модель является развитием модели клиент-сервер (рис. 1.5).

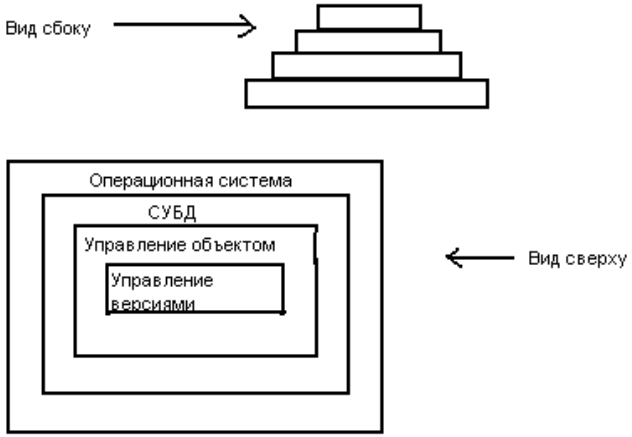

Рис. 1.6. Модель абстрактной машины

Уровень графического интерфейса пользователя запускается на машине клиента. Бизнес-логику образуют модули, осуществляющие функциональные обязанности системы. Этот уровень запускается на сервере приложения. Реляционная СУБД хранит данные, требуемые уровню бизнес-логики. Этот уровень запускается на втором сервере – сервере БД.

Преимущества трёхуровневой модели:

- упрощается такая модификация уровня, которая не влияет на другие уровни;
- происходит отделение прикладных функций от функций управления БД упрощает оптимизацию всей системы.

Модель абстрактной машины отображает многослойную систему (рис. 1.6).

Каждый текущий слой реализуется с использованием средств, обеспечиваемых слоем-фундаментом.

МОДЕЛИРОВАНИЕ УПРАВЛЕНИЯ

Известны два типа моделей управления:

- модель централизованного управления;
- модель событийного управления.

В модели централизованного управления одна подсистема выделяется как системный контроллер. Её обязанности – руководить работой других подсистем. Различают две разновидности моделей централизованного управления: *модель вызов-возврат* (рис. 1.7) и *модель менеджера* (рис. 1.8), которые используются в системах параллельной обработки.

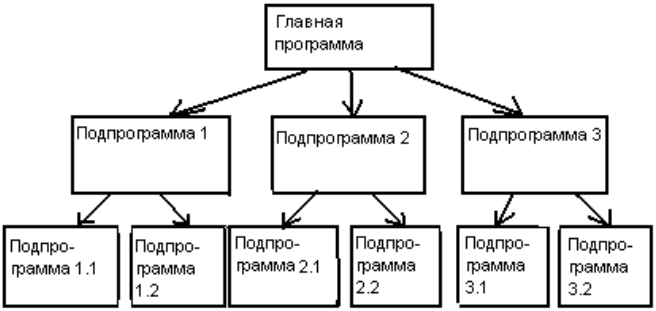

Рис. 1.7. Модель вызов-возврат

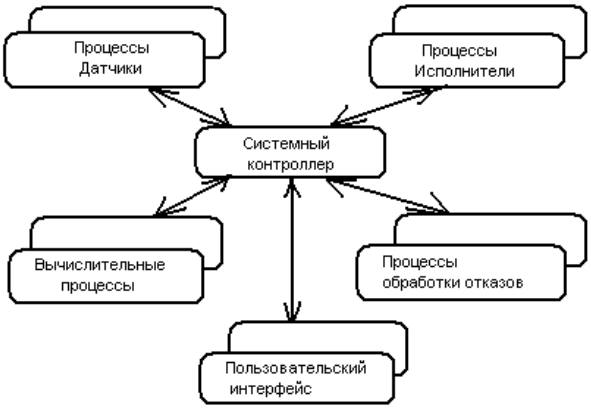

Рис. 1.8. Модель менеджера

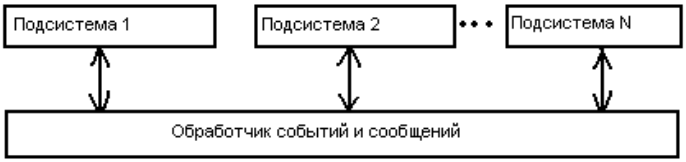

Рис. 1.9. Широковещательная модель

В *широковещательной модели* (рис. 1.9) каждая подсистема уведомляет обработчика о своём интересе к конкретным событиям. Когда событие происходит, обработчик пересылает его подсистеме, которая может обработать это событие. Функции управления в обработчик не встраиваются.

В модели событийного управления системой управляют внешние события. Используются две разновидности модели событийного управления: широковещательная модель и модель, управляемая прерываниями.

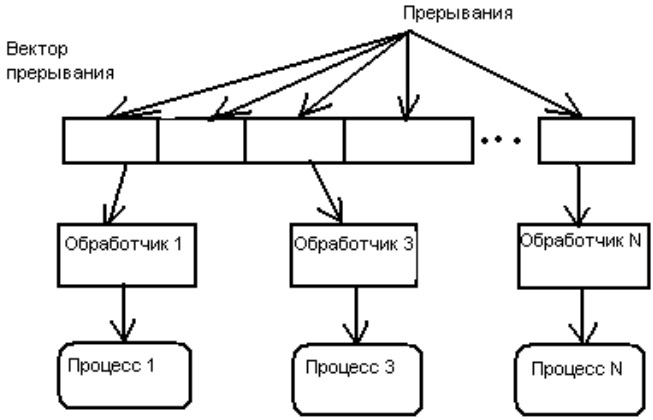

Рис. 1.10. Модель, управляемая прерываниями

В модели, управляемой прерываниями (рис. 1.10), все прерывания разбиты на группы-типы, которые образуют вектор прерываний. Для каждого типа прерывания есть свой обработчик. Каждый обработчик реагирует на свой тип прерывания и запускает свой процесс.

ДЕКОМПОЗИЦИЯ ПОДСИСТЕМ НА МОДУЛИ

Известны два типа моделей модульной декомпозиции:

- модель потока данных;
- модель объектов.

В основе модели потока данных лежит разбиение по функциям.

Модель объектов основана на слабо сцепленных сущностях, имеющих собственные наборы данных, состояния и наборы операций.

Очевидно, что выбор типа декомпозиции должен определяться сложностью разбиваемой подсистемы.

МОДУЛЬНОСТЬ

Модуль – фрагмент программного текста, являющийся строительным блоком для физической структуры системы. Как правило, модуль состоит из интерфейсной части и части-реализации.

Модульность – свойство системы, которая может подвергаться декомпозиции на ряд внутренне связанных и слабо зависящих друг от друга модулей.

По определению Г. Майерса, модульность – свойство ПО, обеспечивающее интеллектуальную возможность создания сколь угодно сложной программы [1]. Проиллюстрируем эту точку зрения.

Пусть $C(x)$ – функция сложности решения проблемы x, $T(x)$ – функция затрат времени на решение проблемы x. Для двух проблем p_1 и p_2 из соотношения $C(p_1) > C(p_2)$ следует, что

$$T(p_1) > T(p_2). \quad (1.1)$$

Этот вывод интуитивно ясен: решение сложной проблемы требует большего времени.

Далее. Из практики решения проблем человеком следует

$$C(p_1 + p_2) > C(p_1) + C(p_2).$$

Отсюда с учётом соотношения (4.1) запишем

$$T(p_1 + p_2) > T(p_1) + T(p_2). \quad (1.2)$$

Соотношение (1.2) – это обоснование модульности. Оно приводит к заключению «разделяй и властвуй» – сложную проблему легче решить, разделив её на управляемые части. Результат, выраженный неравенством (1.2), имеет важное значение для модульности и ПО. Фактически, это аргумент в пользу модульности.

Однако здесь отражена лишь часть реальности, ведь здесь не учитываются затраты на межмодульный интерфейс. Как показано на рис. 1.11, с увеличением количества модулей (и уменьшением их размера) эти затраты также растут.

Таким образом, существует оптимальное количество модулей Opt, которое приводит к минимальной стоимости разработки. Увы, у нас нет необходимого опыта для гарантированного предсказания Opt. Впрочем, разработчики знают, что оптимальный модуль должен удовлетворять двум критериям:

- снаружи он проще, чем внутри;
- его проще использовать, чем построить.

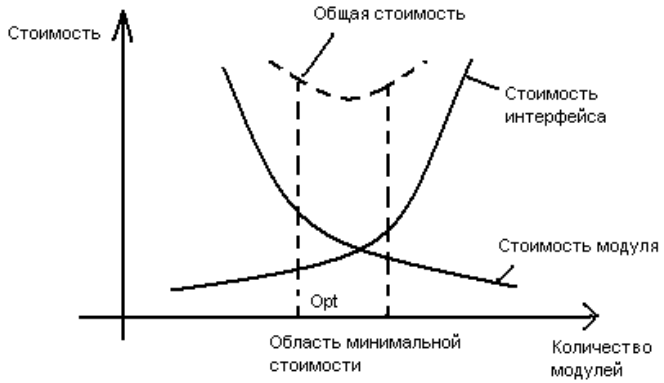

Рис. 1.11. Затраты на модульность

ИНФОРМАЦИОННАЯ ЗАКРЫТОСТЬ

Принцип информационной закрытости утверждает: содержание модулей должно быть скрыто друг от друга [2]. Как показано на рис. 1.12, модуль должен определяться и проектироваться так, чтобы его содержимое (процедуры и данные) было недоступно тем модулям, которые не нуждаются в такой информации (клиентам).

Информационная закрытость означает следующее:

1) все модули независимы, обмениваются только информацией, необходимой для работы;

2) доступ к операциям и структурам данных модуля ограничен.

Достоинства информационной закрытости:

- обеспечивается возможность разработки модулей различными, независимыми коллективами;
- обеспечивается лёгкая модификация системы (вероятность распространения ошибок очень мала, так как большинство данных и процедур скрыто от других частей системы).

Идеальный модуль играет роль «чёрного ящика», содержимое которого невидимо клиентам. Он прост в использовании – количество «ручек и органов управления» им невелико (аналогия с эксплуатацией телевизора). Его легко развивать и корректировать в процессе сопровождения программной системы. Для обеспечения таких возможностей система внутренних и внешних связей модуля должна отвечать особым требованиям. Обсудим характеристики внутренних и внешних связей модуля.

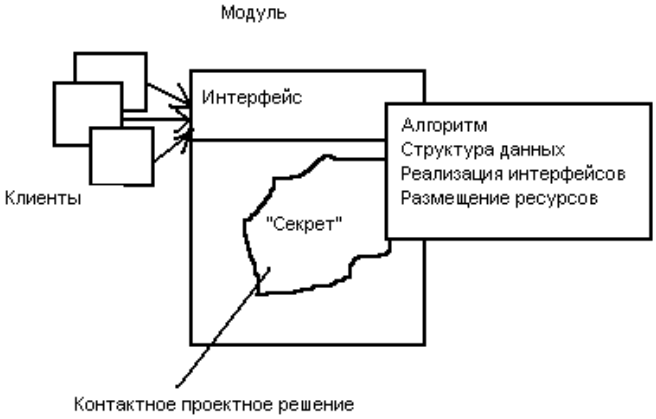

Рис. 1.12. Информационная закрытость модуля

СВЯЗНОСТЬ МОДУЛЯ

Связность модуля (Cohesion) – это мера зависимости его частей [3], [4], [5]. Связность – внутренняя характеристика модуля. Чем выше связность модуля, тем лучше результат проектирования, т.е. тем «черней» его ящик (капсула, защитная оболочка модуля), тем меньше «ручек управления» на нём находится и тем проще эти «ручки».

Для измерения связности используют понятие силы связности (СС). Существует 7 типов связности:

1. **Связность по совпадению** (СС = 0). В модуле отсутствуют явно выраженные внутренние связи.

2. **Логическая связность** (СС = 1). Части модуля объединены по принципу функционального подобия. Например, модуль состоит из разных подпрограмм обработки ошибок. При использовании такого модуля клиент выбирает только одну из подпрограмм.

Недостатки:

- сложное сопряжение;
- большая вероятность внесения ошибок при изменении сопряжения ради одной из функций.

3. **Временная связность** (СС = 3). Части модуля не связаны, но необходимы в один и тот же период работы системы.

Недостаток: сильная взаимная связь с другими модулями, отсюда сильная чувствительность внесению изменений.

4. **Процедурная связность** (СС = 5). Части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения.

5. **Коммуникативная связность** (СС = 7). Части модуля связаны по данным (работают с одной и той же структурой данных).

6. **Информационная (последовательная) связность** (СС = 9). Выходные данные одной части используются как входные данные в другой части модуля.

7. **Функциональная связность** (СС = 10). Части модуля вместе реализуют одну функцию.

1.1. Характеристика связности модуля

Тип связности	Сопровождаемость	Роль модуля
Функциональная	Лучшая сопровождаемость	«Чёрный ящик»
Информационная (последовательная)		Не совсем «чёрный ящик»
Коммуникативная		«Серый ящик»

Продолжение табл. 1.1

Тип связности	Сопровождаемость	Роль модуля
Процедурная	Худшая сопровождаемость	«Белый» или «просвечивающий ящик»
Временная		«Белый ящик»
Логическая		
По совпадению		

Отметим, что типы связности 1, 2, 3 – результат неправильного планирования архитектуры, а тип связности 4 – результат небрежного планирования архитектуры приложения.

Общая характеристика типов связности представлена в табл. 1.1.

ФУНКЦИОНАЛЬНАЯ СВЯЗНОСТЬ

Функционально связный модуль содержит элементы, участвующие в выполнении одной и только одной проблемной задачи. Примеры функционально связных модулей:

- вычислять синус угла;
- проверять орфографию;
- читать запись файла;
- вычислять координаты цели;
- вычислять зарплату сотрудника;
- определять место пассажира.

Каждый из этих модулей имеет единичное назначение. Когда клиент вызывает модуль, выполняется только одна работа, без привлечения внешних обработчиков. Например, модуль «Определять место пассажира» должен делать только это; он не должен распечатывать заголовки страницы.

Некоторые из функционально связных модулей очень просты (например, «Вычислять синус угла» или «Читать запись файла»), другие сложны (например, «Вычислять координаты цели»). Модуль «Вычислять синус угла», очевидно, реализует единичную функцию, но как может модуль «Вычислять зарплату сотрудника» выполнять только одно действие? Ведь каждый знает, что приходится определять начисленную сумму, вычеты по рассрочкам, подоходный налог, социальный налог, алименты и т.д. Дело в том, что несмотря на сложность модуля и на то, что его обязанность исполняют несколько подфункций, если его действия можно представить как единую проблемную функцию (с точки зрения клиента), тогда считают, что модуль функционально связен.

Приложения, построенные из функционально связных модулей, легче всего сопровождать. Напрасно думать, что любой модуль можно рассматривать как однофункциональный. Существует много разновидностей модулей, которые выполняют для клиентов перечень различных работ, и этот перечень нельзя рассматривать как единую проблемную функцию. Критерий при определении уровня связности этих нефункциональных модулей – как связаны друг с другом различные действия, которые они исполняют.

ИНФОРМАЦИОННАЯ СВЯЗНОСТЬ

При информационной (последовательной) связности элементы-обработчики модуля образуют конвейер для обработки данных – результаты одного обработчика используются как исходные данные для следующего обработчика. Приведём пример:

```
модуль «Приём и проверка записи»
    прочитать запись из файла;
    проверить контрольные данные в записи;
    удалить контрольные поля в записи;
    вернуть обработанную запись;
конец модуля.
```

В этом модуле 3 элемента. Результаты первого элемента (прочитать запись из файла) используются как входные данные для второго элемента (проверить контрольные данные в записи) и т.д.

Сопровождать модули с информационной связностью почти так же легко, как и функционально связные модули. Правда, возможности повторного использования здесь ниже, чем в случае функциональной связности. Причина – совместное применение действий модуля с информационной связностью полезно далеко не всегда.

КОММУНИКАТИВНАЯ СВЯЗНОСТЬ

При коммуникативной связности элементы-обработчики модуля используют одни и те же данные, например, внешние данные. Пример коммуникативно связного модуля:

```
модуль «Отчёт и средняя зарплата»
    используется «Таблица зарплаты служащих»;
    сгенерировать «Отчёт по зарплате»;
    вычислить параметр «Средняя зарплата»;
    вернуть «Отчёт по зарплате. Средняя зарплата»;
конец модуля.
```

Здесь все элементы модуля работают со структурой «Таблица зарплаты служащих».

С точки зрения клиента, проблема применения коммуникативно связного модуля состоит в избыточности получаемых результатов. Например, клиенту требуется только отчёт по зарплате, он не нуждается в значении средней зарплаты. Такой клиент будет вынужден выполнять избыточную работу – выделение в полученных данных материала отчёта. Почти всегда разбиение коммуникативно связного модуля на отдельные функционально связные модули улучшает сопровождаемость системы.

Попытаемся провести аналогию между информационной и коммуникативной связностью.

Модули с коммуникативной и информационной связностью схожи в том, что содержат элементы, связанные по данным. Их удобно использовать, потому что лишь немногие элементы в этих модулях связаны с внешней средой. Главное различие между ними – информационно связный модуль работает подобно сборочной линии; его обработчики действуют в определённом порядке; в коммуникативно связном модуле порядок выполнения действий безразличен. В нашем примере не имеет значения, когда генерируется отчёт (до, после или одновременно с вычислением средней зарплаты).

ПРОЦЕДУРНАЯ СВЯЗНОСТЬ

При достижении процедурной связности мы попадаем в пограничную область между хорошей сопровождаемостью (для модулей с более высокими уровнями связности) и плохой сопровождаемостью (для модулей с более низкими уровнями связности). Процедурно связный модуль состоит из элементов, реализующих независимые действия, для которых задан порядок работы, т.е. порядок передачи управления. Зависимости по данным между элементами нет. Например:

```
модуль «Вычисление средних значений»
    используется «Таблица-А». «Таблица-В»;
    вычислить среднее по «Таблица-А»;
    вычислить среднее по «Таблица-В»;
    вернуть среднее «Таблица-А». «Таблица-В»;
конец модуля.
```

Этот модуль вычисляет средние значения для двух, полностью несвязанных таблиц «Таблица-А» и «Таблица-В», каждая из которых имеет по 300 элементов.

Теперь представим себе программиста, которому поручили реализовать данный модуль. Соблазнившись возможностью минимизации кода (использовать один цикл в интересах двух обработчиков, ведь они находятся внутри единого модуля!), программист пишет:

```
модуль «Вычисление средних значений»
    используется «Таблица-А». «Таблица-В»;
    сумма Таблица-А := 0;
    сумма Таблица-В := 0;
    для i := 1 до 300;
    сумма Таблица-А := сумма Таблица-А + Таблица-А(i);
    сумма Таблица-В : = сумма Таблица-В + Таблица-В(i);
    конец для
    среднее Таблица-А := сумма Таблица-А / 300;
    среднее Таблица-В := сумма Таблица-В / 300;
    вернуть среднее Таблица-А, среднее Таблица-В;
конец модуля.
```

Для процедурной связности этот случай типичен – независимый (на уровне проблемы) код стал зависимым (на уровне реализации). Прошли годы, продукт сдали заказчику. И вдруг возникла задача сопровождения – модифицировать модуль под уменьшение размера таблицы В. Оцените, насколько удобно её решать.

ВРЕМЕННАЯ СВЯЗНОСТЬ

При связности по времени элементы-обработчики модуля привязаны к конкретному периоду времени (из жизни программной системы).

Классическим примером временной связности является модуль инициализации:

```
модуль «Инициализировать систему»
    перемотать магнитную ленту 1;
    Счётчик магнитной ленты 1 := 0;
    перемотать магнитную ленту 2;
    Счётчик магнитной ленты 2 := 0;
    Таблица текущих записей : = пробел..пробел;
    Таблица количества записей := 0..0;
    Переключатель 1 : = выкл;
    Переключатель 2 := вкл;
конец модуля.
```

Элементы данного модуля почти не связаны друг с другом (за исключением того, что должны выполняться в определённое время). Они все – часть программы запуска системы. Зато элементы более тесно взаимодействуют с другими модулями, что приводит к сложным внешним связям.

Модуль со связностью по времени испытывает те же трудности, что и процедурно связный модуль. Программист соблазняется возможностью совместного использования кода (действиями, которые связаны только по времени), модуль становится трудно использовать повторно.

Так, при желании инициализировать магнитную ленту 2 в другое время, вы столкнётесь с неудобствами. Чтобы не сбрасывать всю систему, придётся или ввести флажки, указывающие инициализируемую часть, или написать другой код для работы с лентой 2. Оба решения ухудшают сопровождаемость.

Процедурно связные модули и модули с временной связностью очень похожи. Степень их непрозрачности изменяется от темно-серого до светло-серого цвета, так как трудно объявить функцию такого модуля без перечисления её внутренних деталей. Различие между ними подобно различию между информационной и коммуникативной связностью. Порядок выполнения действий более важен в процедурно связных модулях. Кроме того, процедурные модули имеют тенденцию к совместному использованию циклов и ветвлений, а модули с временной связностью чаще содержат более линейный код.

ЛОГИЧЕСКАЯ СВЯЗНОСТЬ

Элементы логически связного модуля принадлежат к действиям одной категории, и из этой категории клиент выбирает выполняемое действие. Рассмотрим следующий пример:

```
модуль «Пересылка сообщения»
    переслать по электронной почте;
    переслать по факсу;
    послать в телеконференцию;
    переслать по ftp-протоколу;
конец модуля.
```

Как видим, логически связный модуль – мешок доступных действий. Действия вынуждены совместно использовать один и тот же интерфейс модуля. В строке вызова модуля значение каждого параметра зависит от используемого действия. При вызове отдельных действий некоторые параметры должны иметь значение пробела, нулевые значения и т.д. (хотя клиент всё же должен использовать их и знать их типы).

Действия в логически связном модуле попадают в одну категорию, хотя имеют не только сходства, но и различия. К сожалению, это заставляет программиста «завязывать код действий в узел», ориентируясь на то, что действия совместно используют общие строки кода. Поэтому логически связный модуль имеет:

- уродливый внешний вид с различными параметрами, обеспечивающими, например, четыре вида доступа;
- запутанную внутреннюю структуру со множеством переходов, похожую на волшебный лабиринт.

В итоге модуль становится сложным как для понимания, так и для сопровождения.

СВЯЗНОСТЬ ПО СОВПАДЕНИЮ

Элементы связного по совпадению модуля вообще не имеют никаких отношений друг с другом:

```
модуль «Разные функции» (какие-то параметры)
    поздравить с Новым годом (...);
    проверить исправность аппаратуры (...);
    заполнить анкету героя (...);
    измерить температуру (...);
    вывести собаку на прогулку (...);
    запастись продуктами (...);
    приобрести «ягуар» (...);
конец модуля.
```

Связный по совпадению модуль похож на логически связный модуль. Его элементы-действия не связаны ни потоком данных, ни потоком управления. Но в логически связном модуле действия, по крайней мере, относятся к одной категории; в связном по совпадению модуле даже это не так. Словом, связные по совпадению модули имеют все недостатки логически связных модулей и даже усиливают их. Применение таких модулей вселяет ужас, поскольку один параметр используется для разных целей.

Чтобы клиент мог воспользоваться модулем «Разные функции», этот модуль (подобно всем связным по совпадению модулям) должен быть «белым ящиком», чья реализация полностью видима. Такие модули делают системы менее понятными и труднее сопровождаемыми, чем системы без модульности вообще.

К счастью, связность по совпадению встречается редко. Среди её причин можно назвать:

- бездумный перевод существующего монолитного кода в модули;
- необоснованные изменения модулей с плохой (обычно временной) связностью, приводящие к добавлению флажков.

ОПРЕДЕЛЕНИЕ СВЯЗНОСТИ МОДУЛЯ

Приведём алгоритм определения уровня связности модуля.

1. Если модуль – единичная проблемно-ориентированная функция, то уровень связности – функциональный; конец алгоритма. В противном случае перейти к пункту 2.

2. Если действия внутри модуля связаны, то перейти к пункту 3. Если действия внутри модуля никак не связаны, то перейти к пункту 6.

3. Если действия внутри модуля связаны данными, то перейти к пункту 4. Если действия внутри модуля связаны потоком управления, перейти к пункту 5.

4. Если порядок действий внутри модуля важен, то уровень связности – информационный. В противном случае уровень связности – коммуникативный. Конец алгоритма.

5. Если порядок действий внутри модуля важен, то уровень связности – процедурный. В противном случае уровень связности – временной. Конец алгоритма.

6. Если действия внутри модуля принадлежат к одной категории, то уровень связности – логический. Если действия внутри модуля не принадлежат к одной категории, то уровень связности – по совпадению. Конец алгоритма.

Возможны более сложные случаи, когда с модулем ассоциируются несколько уровней связности. В этих случаях следует применять одно из двух правил:

- правило параллельной цепи. Если все действия модуля имеют несколько уровней связности, то модулю присваивают самый сильный уровень связности;
- правило последовательной цепи. Если действия в модуле имеют разные уровни связности, то модулю присваивают самый слабый уровень связности.

Например, модуль может содержать некоторые действия, которые связаны процедурно, а также другие действия, связные по совпадению. В этом случае применяют правило последовательной цепи, и в целом, модуль считают связным по совпадению.

СЦЕПЛЕНИЕ МОДУЛЕЙ

Сцепление (Coupling) – мера взаимозависимости модулей по данным [3], [4], [5]. Сцепление – внешняя характеристика модуля, которую желательно уменьшать.

Количественно сцепление измеряется степенью сцепления (СЦ). Выделяют шесть типов сцепления.

1. **Сцепление по данным** (СЦ = 1). Модуль А вызывает модуль В.

Все входные и выходные параметры вызываемого модуля – простые элементы данных (рис. 1.13).

2. **Сцепление по образцу** (СЦ = 3). В качестве параметров используются структуры данных (рис. 1.14).

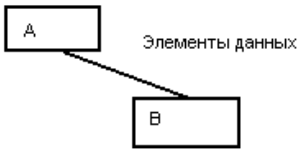

Рис. 1.13. Сцепление по данным

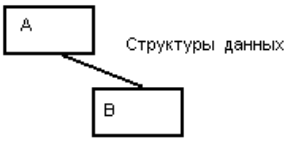

Рис. 1.14. Сцепление по образцу

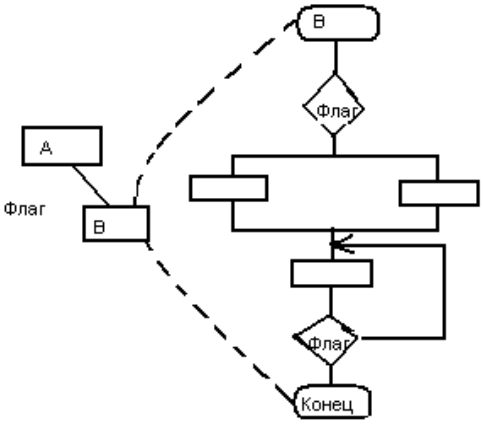

Рис. 1.15. Сцепление по управлению

3. **Сцепление по управлению** (СЦ = 4). Модуль A явно управляет функционированием модуля B (с помощью флагов или переключателей), посылая ему управляющие данные (рис. 1.15).

4. **Сцепление по внешним ссылкам** (СЦ = 5). Модули A и B ссылаются на один и тот же глобальный элемент данных.

5. **Сцепление по общей области** (СЦ = 7). Модули разделяют одну и ту же глобальную структуру данных (рис. 1.16).

6. **Сцепление по содержанию** (СЦ = 9). Один модуль прямо ссылается на содержание другого модуля (не через его точку входа). Например, коды их команд перемежаются друг с другом (рис. 1.16).

На рисунке 1.16 видим, что модули B и D сцеплены по содержанию, а модули C, E и N сцеплены по общей области.

Рис. 1.16. Сцепление по общей области и содержанию

СЛОЖНОСТЬ ПРОГРАММНОЙ СИСТЕМЫ

В простейшем случае сложность системы определяется как сумма мер сложности её модулей. Сложность модуля может вычисляться различными способами.

Например, в [6] предложена мера длины N модуля

$$N \approx n_1 \log_2(n_1) + n_2 \log_2(n_2),$$

где n_1 – число различных операторов; n_2 – число различных операндов.

В качестве второй метрики М. Холстед рассматривал объём V модуля (количество символов для записи всех операторов и операндов текста программы):

$$V = N \times \log_2(n_1 + n_2).$$

Вместе с тем известно, что любая сложная система состоит из элементов и системы связей между элементами и что игнорировать внутрисистемные связи неразумно.

При оценке сложности ПС в [7] предложено исходить из топологии внутренних связей. Для этой цели он разработал метрику цикломатической сложности

$$V(G) = E - N + 2,$$

где E – количество дуг; N – количество вершин в управляющем графе ПС.

Это был шаг в нужном направлении. Дальнейшее уточнение оценок сложности потребовало, чтобы каждый модуль мог представляться как локальная структура, состоящая из элементов и связей между ними.

Таким образом, при комплексной оценке сложности ПС необходимо рассматривать меру сложности модулей, меру сложности внешних связей (между модулями) и меру сложности внутренних связей (внутри модулей) [8], [9]. Традиционно со внешними связями сопоставляют характеристику «сцепление», а с внутренними связями – характеристику «связность».

Вопросы комплексной оценки сложности обсудим в следующем разделе.

ХАРАКТЕРИСТИКИ ИЕРАРХИЧЕСКОЙ СТРУКТУРЫ ПРОГРАММНОЙ СИСТЕМЫ

Иерархическая структура программной системы – основной результат предварительного проектирования. Она определяет состав модулей ПС и управляющие отношения между модулями. В этой структуре модуль более высокого уровня (начальник) управляет модулем нижнего уровня (подчинённым).

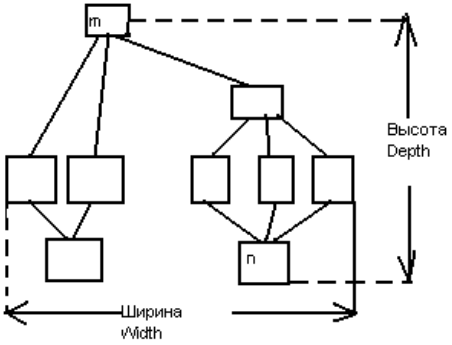

Рис. 1.17. Иерархическая структура программной системы

Иерархическая структура не отражает процедурные особенности программной системы, т.е. последовательность операций, их повторение, ветвление и т.д. Рассмотрим основные характеристики иерархической структуры, представленной на рис. 1.17.

Первичными характеристиками являются количество вершин (модулей) и количество рёбер (связей между модулями). К ним добавляются две глобальные характеристики – высота и ширина:

- **высота** – количество уровней управления;
- **ширина** – максимальное из количеств модулей, размещённых на уровнях управления.

В нашем примере высота = 4, ширина = 6.

Локальными характеристиками модулей структуры являются коэффициент объединения по входу и коэффициент разветвления по выходу.

Коэффициент объединения по входу Fan_in(i) – это количество модулей, которые прямо управляют i-м модулем.

В примере для модуля n: Fan_in(n)=4.

Коэффициент разветвления по выходу Fan_out(i) – это количество модулей, которыми прямо управляет i-й модуль.

В примере для модуля m: Fan_out(m)=3.

Возникает вопрос: как оценить качество структуры? Из практики проектирования известно, что лучшее решение обеспечивается иерархической структурой в виде дерева.

Степень отличия реальной проектной структуры от дерева характеризуется невязкой структурой. Как определить невязку?

Вспомним, что полный граф (complete graph) с n вершинами имеет количество рёбер

$$e_c = n(n-1)/2,$$

а дерево (tree) с таким же количеством вершин – существенно меньшее количество рёбер

$$e_t = n - 1.$$

Тогда формулу невязки можно построить, сравнивая количество рёбер полного графа, реального графа и дерева.

Для проектной структуры с n вершинами и e рёбрами невязка определяется по выражению

$$\text{Nev} = \frac{e - e_t}{e_c - e_t} = \frac{(e - n + 1)2}{n(n-1) - 2(n-1)} = \frac{2(e - n + 1)}{(n-1)\,(n-2)}.$$

Значение невязки лежит в диапазоне от 0 до 1. Если Nev = 0, то проектная структура является деревом, если Nev = 1, то проектная структура – полный граф.

Ясно, что невязка даёт грубую оценку структуры. Для увеличения точности оценки следует применить характеристики связности и сцепления.

Хорошая структура должна иметь низкое сцепление и высокую связность.

В [5] предложено оценивать структуру с помощью коэффициентов Fan_in(i) и Fan_out(i) модулей.

Большое значение Fan_in(i) – свидетельство высокого сцепления, так как является мерой зависимости модуля. Большое значение Fan_out(i) говорит о высокой сложности вызывающего модуля. Причиной является то, что для координации подчинённых модулей требуется сложная логика управления.

Основной недостаток коэффициентов Fan_in(i) и Fan_out(i) состоит в игнорировании веса связи. Здесь рассматриваются только управляющие потоки (вызовы модулей). В то же время информационные потоки, нагружающие рёбра структуры могут существенно изменяться, поэтому нужна мера, которая учитывает не только количество рёбер, но и количество информации, проходящей через них.

В [10] введены информационные коэффициенты ifan_in(i) и ifan_out(j). Они учитывают количество элементов и структур данных, из которых i-й модуль берёт информацию и которые обновляются j-м модулем соответственно.

Информационные коэффициенты суммируются со структурными коэффициентами sfan_in(i) и sfan_out(j), которые учитывают только вызовы модулей.

В результате формируются полные значения коэффициентов:

$$\text{Fan\_in}\ (i) = \text{sfan\_in}\ (i) + \text{ifan\_in}\ (i),$$

$$\text{Fan\_out}\ (j) = \text{sfan\_out}\ (j) + \text{ifan\_out}\ (j).$$

На основе полных коэффициентов модулей вычисляется метрика общей сложности структуры

$$S = \sum_{i=1}^{n} \text{length}(i) \times (\text{Fan\_in}(i) + \text{Fan\_out}(i))^2,$$

где length(i) – оценка размера i-го модуля (в виде LOC- или FP-оценки).

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какова цель синтеза программной системы? Перечислите этапы синтеза.
2. Дайте определение разработки данных, разработки архитектуры и процедурной разработки.
3. Какие особенности имеет этап проектирования?
4. Решение каких задач обеспечивает предварительное проектирование?
5. Какие модели системного структурирования вы знаете?
6. Чем отличается модель клиент-сервер от трёхуровневой модели?
7. Какие типы моделей управления вы знаете?
8. Какие существуют разновидности моделей централизованного управления?
9. Поясните разновидности моделей событийного управления.
10. Поясните понятия модуля и модульности. Зачем используют модули?
11. В чём состоит принцип информационной закрытости? Какие достоинства он имеет?
12. Что такое связность модуля?
13. Какие существуют типы связности?
14. Дайте характеристику функциональной связности.
15. Дайте характеристику информационной связности.
16. Охарактеризуйте коммуникативную связность.
17. Охарактеризуйте процедурную связность.
18. Дайте характеристику временной связности.
19. Дайте характеристику логической связности.
20. Охарактеризуйте связность по совпадению.
21. Что значит «улучшать связность»?
22. Что такое сцепление модуля?
23. Какие существуют типы сцепления?
24. Дайте характеристику сцепления по данным.
25. Дайте характеристику сцепления по образцу.
26. Охарактеризуйте сцепление по управлению.
27. Охарактеризуйте сцепление по внешним ссылкам.

28. Дайте характеристику сцепления по общей области.
29. Дайте характеристику сцепления по содержанию.
30. Что значит «улучшать сцепление»?
31. Какие подходы к оценке сложности системы вы знаете?
32. Что определяет иерархическая структура программной системы?
33. Поясните первичные характеристики иерархической структуры.
34. Поясните понятия коэффициента объединения по входу и коэффициента разветвления по выходу.
35. Что определяет невязка структуры?
36. Поясните информационные коэффициенты объединения и разветвления.

2. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРЕДСТАВЛЕНИЯ ПРОГРАММНЫХ СИСТЕМ

ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРЕДСТАВЛЕНИЯ ПРОГРАММНЫХ СИСТЕМ

Рассмотрение любой сложной системы требует применения техники декомпозиции – разбиения на составляющие элементы. Известны две схемы декомпозиции: алгоритмическая декомпозиция и объектно-ориентированная декомпозиция.

В основе алгоритмической декомпозиции лежит разбиение по действиям – алгоритмам. Эта схема представления применяется в обычных ПС.

Объектно-ориентированная декомпозиция обеспечивает разбиение по автономным лицам – объектам реального (или виртуального) мира. Эти лица (объекты) – более «крупные» элементы, каждый из них несёт в себе и описания действий, и описания данных.

Объектно-ориентированное представление ПС основывается на принципах абстрагирования, инкапсуляции, модульности и иерархической организации. Каждый из этих принципов не нов, но их совместное применение рассчитано на проведение объектно-ориентированной декомпозиции. Это определяет модификацию их содержания и механизмов взаимодействия друг с другом. Обсудим данные принципы [11], [12], [13], [14], [15], [16].

АБСТРАГИРОВАНИЕ

Аппарат абстракции – удобный инструмент для борьбы со сложностью реальных систем. Создавая понятие в интересах какой-либо задачи, мы отвлекаемся (абстрагируемся) от несущественных характеристик конкретных объектов, определяя только существенные характеристики. Например, в абстракции «часы» мы выделяем характеристику «показывать время», отвлекаясь от таких характеристик конкретных часов, как форма, цвет, материал, цена, изготовитель.

Итак, абстрагирование сводится к формированию абстракций. Каждая абстракция фиксирует основные характеристики объекта, которые отличают его от других видов объектов и обеспечивают ясные понятийные границы.

Абстракция концентрирует внимание на внешнем представлении объекта, позволяет отделить основное в поведении объекта от его реализации. Абстракцию удобно строить путём выделения обязанностей объекта.

Пример: физический объект – датчик скорости, устанавливаемый на борту летательного аппарата (ЛА). Создадим его абстракцию. Для этого сформулируем обязанности датчика:

- знать проекцию скорости ЛА в заданном направлении;
- показывать текущую скорость;
- подвергаться настройке.

Теперь опишем абстракцию датчика. Описание сформулируем как спецификацию класса:

```
класс «ДатчикСкорости» это
    поле «Скорость вещественный» ...;
    поле «Направление строковый» ...;
    тип «ДатчикСкорости» закрытый;
    функция «НовыйДатчик»(Направление);
    вернуть «ДатчикСкорости»;
    функция «ТекущаяСкорость»(ДатчикСкорости)
    вернуть Скорость;
    процедура «Настраивать»(ДатчикСкорости;
    ДействительнаяСкорость: Скорость);
класс «ДатчикСкорости».
```

Здесь Скорость и Направление – вспомогательные подтипы, обеспечивающие задание операций абстракции (НовыйДатчик, ТекущаяСкорость, Настраивать). Приведённая абстракция – это только спецификация класса датчика, настоящее его представление скрыто в приватной части спецификации и теле класса. Класс «ДатчикСкорости» – ещё не объект. Собственно датчики – это его экземпляры, и их нужно создать, прежде чем с ними можно будет работать. Например, можно написать так:

```
ДатчикПродольнойСкорости : ДатчикСкорости;
ДатчикПоперечнойСкорости : ДатчикСкорости;
ДатчикНормальнойСкорости : ДатчикСкорости.
```

ИНКАПСУЛЯЦИЯ

Инкапсуляция и абстракция – взаимодополняющие понятия: абстракция выделяет внешнее поведение объекта, а инкапсуляция содержит и скрывает реализацию, которая обеспечивает это поведение. Инкапсуляция достигается с помощью информационной закрытости. Обычно скрываются структура объектов и реализация их методов.

Инкапсуляция является процессом разделения элементов абстракции на секции с различной видимостью. Инкапсуляция служит для отделения интерфейса абстракции от её реализации.

П р и м е р: физический объект «Регулятор скорости».
Обязанности регулятора:

- включаться;
- выключаться;
- увеличивать скорость;
- уменьшать скорость;
- отображать своё состояние.

Спецификация класса «Регулятор скорости» примет вид:

```
класс «ДатчикСкорости». Класс «Порт»;
использовать Класс «ДатчикСкорости». Класс «Порт»;
класс «РегуляторСкорости» это
    поле «Режим тип» (Увеличение, Уменьшение);
    поле «Размещение строковый»;
    поле «РегуляторСкорости» закрытое;
    фнукция «НовыйРегуляторСкорости» (номер: Размещение;
        например: Направление; Порт: Порт);
        вернуть «РегуляторСкорости»;
    процедура «Включить»(РегуляторСкорости);
    процедура «Выключить»(РегуляторСкорости);
    процедура «УвеличитьСкорость»(РегуляторСкорости);
    процедура «УменьшитьСкорость»(РегуляторСкорости);
    функция «ОпросСостояния»(РегуляторСкорости);
закрытый
    поле указатель на Порт доступ Порт;
    поле «РегуляторСкорости» запись
        Номер; Размещение;
        Состояние; Режим;
        Управление: указатель на Порт;
    конец записи;
конец Класс «РегуляторСкорости».
```

Здесь вспомогательный тип «Режим» используется для задания основного типа класса, класс «ДатчикСкорости» обеспечивает класс регулятора описанием вспомогательного типа «Направление», класс «Порт» фиксирует абстракцию порта, через который посылаются сообщения для регулятора. Три свойства: Номер, Состояние, Управление – формулируют инкапсулируемое представление основного типа класса «РегуляторСкорости». При попытке клиента получить доступ к этим свойствам фиксируется семантическая ошибка.

Полное инкапсулированное представление класса «РегуляторСкорости» включает описание реализаций его методов – оно содержится в теле класса. Описание тела для краткости здесь опущено.

МОДУЛЬНОСТЬ

В языках C++, Object Pascal абстракции классов и объектов формируют логическую структуру системы. При производстве физической структуры эти абстракции помещаются в модули. В больших системах, где классов сотни, модули помогают управлять сложностью. Модули служат физическими контейнерами, в которых объявляются классы и объекты логической разработки.

Модульность определяет способность системы подвергаться декомпозиции на ряд сильно связанных и слабо сцепленных модулей.

Общая цель декомпозиции на модули: уменьшение сроков разработки и стоимости ПС за счёт выделения модулей, которые проектируются и изменяются независимо. Каждая модульная структура должна быть достаточно простой, чтобы быть полностью понятой. Изменение реализации модулей должно проводиться без знания реализации других модулей и без влияния на их поведение.

Определение классов и объектов выполняется в ходе логической разработки, а определение модулей – в ходе физической разработки системы. Эти действия сильно взаимосвязаны, осуществляются итеративно.

ИЕРАРХИЧЕСКАЯ ОРГАНИЗАЦИЯ

Мы рассмотрели три механизма для борьбы со сложностью:

- абстракцию (она упрощает представление физического объекта);
- инкапсуляцию (закрывает детали внутреннего представления абстракций);
- модульность (даёт путь группировки логически связанных абстракций).

Прекрасным дополнением к этим механизмам является иерархическая организация – формирование из абстракций иерархической структуры. Определением иерархии в проекте упрощаются понимание проблем заказчика и их реализация – сложная система становится обозримой человеком.

Иерархическая организация задаёт размещение абстракций на различных уровнях описания системы.

Двумя важными инструментами иерархической организации в объектно-ориентированных системах являются:

- структура из классов («*is a*»-иерархия);
- структура из объектов («*part of*»-иерархия).

Чаще всего «is a»-иерархическая структура строится с помощью наследования. Наследование определяет отношение между классами, где класс разделяет структуру или поведение, определённые в одном другом (единичное наследование) или в нескольких других (множественное наследование) классах.

Другая разновидность иерархической организации – «part of»-иерархическая структура – базируется на отношении агрегации. Агрегация не является понятием, уникальным для объектно-ориентированных систем. Например, любой язык программирования, разрешающий структуры типа «запись», поддерживает агрегацию. И всё же агрегация особенно полезна в сочетании с наследованием:

1) агрегация обеспечивает физическую группировку логически связанной структуры;

2) наследование позволяет легко и многократно использовать эти общие группы в других абстракциях.

Интересно сравнить элементы иерархий наследования и агрегации с точки зрения уровня сложности. При наследовании нижний элемент иерархии (подкласс) имеет больший уровень сложности (большие возможности), при агрегации – наоборот (агрегат ИзмерительСУ обладает большими возможностями, чем его элементы – датчики и процедура настройки).

ОБЪЕКТЫ

Рассмотрим более пристально объекты – конкретные сущности, которые существуют во времени и пространстве.

ОБЩАЯ ХАРАКТЕРИСТИКА ОБЪЕКТОВ

Объект – это конкретное представление абстракции. Объект обладает индивидуальностью, состоянием и поведением. Структура и поведение подобных объектов определены в их общем классе. Термины «экземпляр класса» и «объект» взаимозаменяемы. На рисунке 2.1 приведён пример объекта по имени Стул, имеющего определённый набор свойств и операций.

Индивидуальность – это характеристика объекта, которая отличает его от всех других объектов.

Состояние объекта характеризуется перечнем всех свойств объекта и текущими значениями каждого из этих свойств (рис. 2.1).

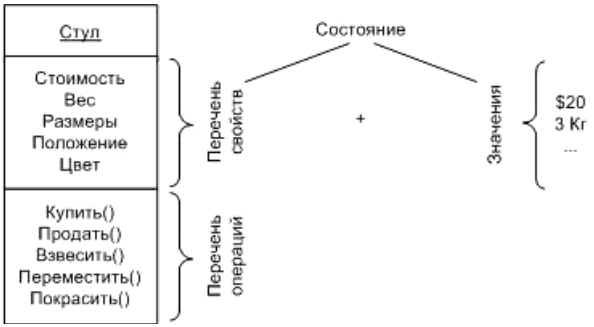

Рис. 2.1. Представление объекта с именем Стул

Объекты не существуют изолированно друг от друга. Они подвергаются воздействию или сами воздействуют на другие объекты.

Поведение характеризует то, как объект воздействует на другие объекты (или подвергается воздействию) в терминах изменений его состояния и передачи сообщений. Поведение объекта является функцией как его состояния, так и выполняемых им операций (Купить, Продать, Взвесить, Переместить, Покрасить). Говорят, что состояние объекта представляет суммарный результат его поведения.

Операция обозначает обслуживание, которое объект предлагает своим клиентам. Возможны пять видов операций клиента над объектом:

1) модификатор (изменяет состояние объекта);
2) селектор (даёт доступ к состоянию, но не изменяет его);
3) итератор (доступ к содержанию объекта по частям в строго определённом порядке);
4) конструктор (создаёт объект и инициализирует его состояние);
5) деструктор (разрушает объект и освобождает занимаемую им память).

Примеры операций приведены в табл. 2.1.

В чистых объектно-ориентированных языках программирования операции могут объявляться только как методы – элементы классов, экземплярами которых являются объекты. Гибридные языки (C++) позволяют писать операции как свободные подпрограммы (вне классов). Соответствующие примеры показаны на рис. 2.2.

2.1. Разновидности операций

Вид операции	Пример операции
Модификатор	Пополнеть (кг)
Селектор	КакойВес ( ) : integer
Итератор	ПоказатьАссортиментТоваров ( ) : string
Конструктор	СоздатьРобот (параметры)
Деструктор	УничтожитьРобот ( )

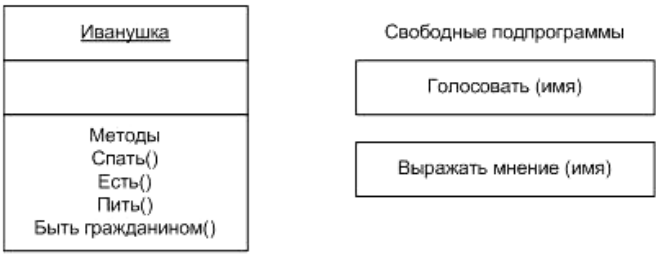

Рис. 2.2. Методы и свободные подпрограммы

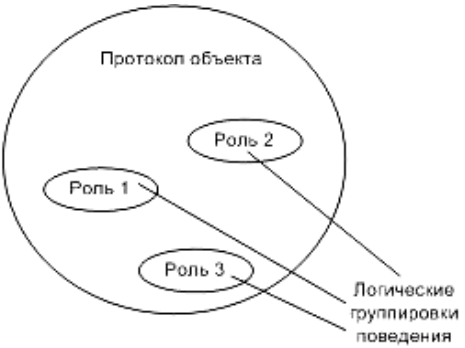

Рис. 2.3. Пространство поведения объекта

В общем случае все методы и свободные подпрограммы, ассоциированные с конкретным объектом, образуют его *протокол*. Таким образом, протокол определяет оболочку допустимого поведения объекта и поэтому заключает в себе цельное (статическое и динамическое) представление объекта.

Большой протокол полезно разделять на логические группировки поведения. Эти группировки, разделяющие пространство поведения объекта, обозначают *роли*, которые может играть объект. Принцип выделения ролей иллюстрирует рис. 2.3.

С точки зрения внешней среды важное значение имеет такое понятие, как обязанности объекта. *Обязанности* означают обязательства объекта обеспечить определённое поведение. Обязанностями объекта являются все виды обслуживания, которые он предлагает клиентам. В мире объект играет определённые роли, выполняя свои обязанности.

В заключение отметим: наличие у объекта внутреннего состояния означает, что порядок выполнения им операций очень важен. Иначе говоря, объект может представляться как независимый автомат. По аналогии с автоматами можно выделять активные и пассивные объекты (рис. 2.4).

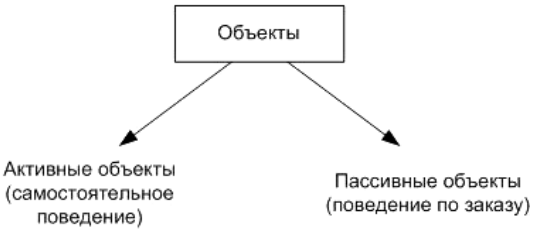

Рис. 2.4. Активные и пассивные объекты

Активный объект имеет собственный канал (поток) управления, пассивный – нет. Активный объект автономен, он может проявлять своё поведение без воздействия со стороны других объектов. Пассивный объект, наоборот, может изменять своё состояние только под воздействием других объектов.

ВИДЫ ОТНОШЕНИЙ МЕЖДУ ОБЪЕКТАМИ

В поле зрения разработчика ПО находятся не объекты-одиночки, а взаимодействующие объекты, ведь именно взаимодействие объектов реализует поведение системы. У Г. Буча есть отличная цитата из Галла: «Самолет – это набор элементов, каждый из которых по своей природе стремится упасть на землю, но ценой совместных непрерывных усилий преодолевает эту тенденцию» [11]. Отношения между парой объектов основываются на взаимной информации о разрешённых операциях и ожидаемом поведении. Интересны два вида отношений между объектами: связи и агрегация.

СВЯЗИ

Связь – это физическое или понятийное соединение между объектами. Объект сотрудничает с другими объектами через соединяющие их связи. Связь обозначает соединение, с помощью которого:

- объект-клиент вызывает операции объекта-поставщика;
- один объект перемещает данные к другому объекту.

Можно сказать, что связи являются рельсами между станциями-объектами, по которым ездят «трамвайчики сообщений».

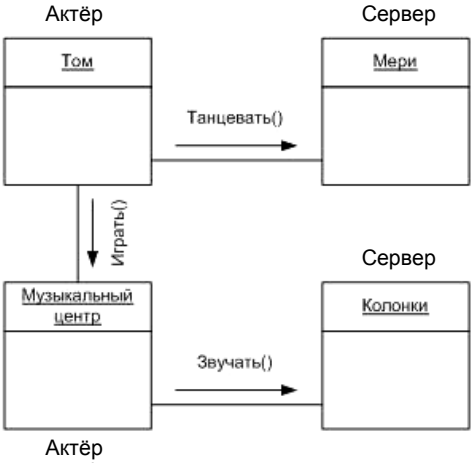

Рис. 2.5. Связи между объектами

Связи между объектами показаны на рис. 2.5 с помощью соединительных линий. Связи представляют возможные пути для передачи сообщений. Сами сообщения показаны стрелками, отмечающими их направления, и помечены именами вызываемых операций.

Как участник связи, объект может играть одну из трёх ролей:

- актёр – объект, который может воздействовать на другие объекты, но никогда не подвержен воздействию других объектов;
- сервер – объект, который никогда не воздействует на другие объекты, он только используется другими объектами;
- агент – объект, который может как воздействовать на другие объекты, так и использоваться ими. Агент создаётся для выполнения работы от имени актёра или другого агента.

На рисунке 2.5, Том – это актёр, Мери, Колонки – серверы, Музыкальный центр – агент.

Приведём пример. Допустим, что нужно обеспечить следующий график разворота первой ступени ракеты по углу тангажа, представленный на рис. 2.6.

Запишем абстракцию графика разворота:

```
with Класс «ДатчикУглаТангажа»;
use Класс «ДатчикУглаТангажа»;
Package Класс «ГрафикРазворота» is
    subtype Секунда is Natural range ...;
    type ГрафикРазворота is tagged private;
    procedure «Очистить» (in out ГрафикРазворота);
    procedure «Связать» (in out ГрафикРазворота;
        teta: Угол: si: Секунда: s2: Секунда);
    function «УголНаМомент» (ГрафикРазворота;
        s: Секунда) return Угол;
private
…
end Класс «ГрафикРазворота».
```

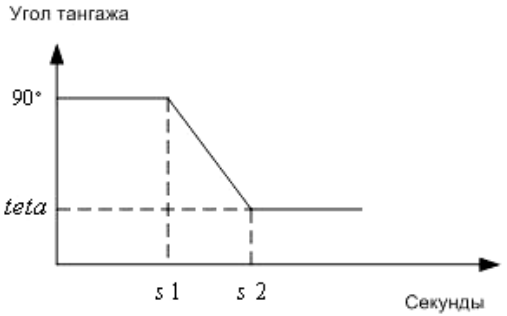

Рис. 2.6. График разворота первой ступени ракеты

Для решения задачи надо обеспечить сотрудничество трёх объектов: экземпляра класса «ГрафикРазворота», «РегуляторУгла» и «КонтроллерУгла».

Описание класса «КонтроллерУгла» может иметь следующий вид:

```
with Класс «ГрафикРазворота». Класс «РегуляторУгла»;
use Класс «ГрафикРазворота». Класс «РегуляторУгла»;
Package Класс «КонтроллерУгла» is
    type указатель на График is access all ГрафикРазворота;
    type «КонтроллерУгла» is tagged private;
    procedure «Обрабатывать» (in out КонтроллерУгла;
        угол: указатель на График);
    function Запланировано (КонтроллерУгла;
        угол: указатель на График) return Секунда;
private
    type «КонтроллерУгла» is tagged record;
    регулятор: РегуляторУгла := Новый РегуляторУгла (1.1.10);
    …
end Класс «КонтроллерУгла».
```

Примечание. Операция «Запланировано» позволяет клиентам запросить у экземпляра «КонтроллерУгла» время обработки следующего графика.

И наконец, описание класса «РегуляторУгла» представим в следующей форме:

```
with Класс «ДатчикУгла». Класс «Порт»;
use Класс «ДатчикУгла». Класс «Порт»;
Package Класс «РегуляторУгла» is
    type Режим is (Увеличение. Уменьшение);
    subtype Размещение is Natural range ...;
    type РегуляторУгла is tagged private;
    function Новый РегуляторУгла (номер: Размещение;
        например: Направление: Порт: Порт)
        return «РегуляторУгла»;
    procedure «Включить»(in out РегуляторУгла);
    procedure «Выключить»(in out РегуляторУгла);
    procedure «УвеличитьУгол»(in out РегуляторУгла);
    procedure «УменьшитьУгол»(in out РегуляторУгла);
    function ОпросСостояния(РегуляторУгла)
        return Режим:
```

```
private
    type указатель на Порт is access all Порт;
    type РегуляторУгла is tagged record;
        Номер: Размещение;
        Состояние: Режим;
        Управление: указатель на Порт;
    end record;
end Класс «РегуляторУгла».
```

Теперь, когда сделаны необходимые приготовления, объявим нужные экземпляры классов, т.е. объекты:

```
РабочийГрафик: aliased ГрафикРазворота;
РабочийКонтроллер: aliased КонтроллерУгла.
```

Далее мы должны определить конкретные параметры графика разворота:

```
связать (РабочийГрафик. 30. 60. 90);
```

а затем предложить объекту-контроллеру выполнить этот график:

```
обрабатывать (РабочийКонтроллер. РабочийГрафикAccess);
```

Рассмотрим отношение между объектом «РабочийГрафик» и объектом «РабочийКонтроллер». РабочийКонтроллер – это агент, отвечающий за выполнение графика разворота и поэтому использующий объект «РабочийГрафик» как сервер. В данном отношении объект «РабочийКонтроллер» использует объект «РабочийГрафик» как аргумент в одной из своих операций.

ВИДИМОСТЬ ОБЪЕКТОВ

Рассмотрим два объекта, A и B, между которыми имеется связь. Для того, чтобы объект A мог послать сообщение в объект B, надо, чтобы B был виден для A.

В примере из предыдущего подраздела объект «РабочийКонтроллер» должен видеть объект «РабочийГрафик» (чтобы иметь возможность использовать его как аргумент в операции «Обрабатывать»).

Различают четыре формы видимости между объектами.

1. Объект-поставщик (сервер) глобален для клиента.
2. Объект-поставщик (сервер) является параметром операции клиента.
3. Объект-поставщик (сервер) является частью объекта-клиента.
4. Объект-поставщик (сервер) является локально объявленным объектом в операции клиента.

На этапе анализа вопросы видимости обычно опускают. На этапах проектирования и реализации вопросы видимости по связям обязательно должны рассматриваться.

АГРЕГАЦИЯ

Связи обозначают равноправные (клиент-серверные) отношения между объектами. Агрегация обозначает отношения объектов в иерархии «целое/часть». Агрегация обеспечивает возможность перемещения от целого (агрегата) к его частям (свойствам).

В примере из подраздела «Связи» объект «РабочийКонтроллер» имеет свойство регулятор, чьим классом является «РегуляторУгла». Поэтому объект «РабочийКонтроллер» является агрегатом (целым), а экземпляр «РегуляторУгла» – одной из его частей. Из объекта «РабочийКонтроллер» всегда можно попасть в объект «РегуляторУгла». Обратный же переход (из части в целое) обеспечивается не всегда.

Агрегация может обозначать, а может и не обозначать физическое включение части в целое. На рисунке 2.7 приведён пример физического включения (композиции) частей (Двигателя, Сидений, Колес) в агрегат Автомобиль. В этом случае говорят, что части включены в агрегат по величине.

На рисунке 2.8 приведён пример нефизического включения частей (Студента, Преподавателя) в агрегат ВУЗ. Очевидно, что Студент и Преподаватель являются элементами ВУЗа, но они не входят в него физически. В этом случае говорят, что части включены в агрегат по ссылке.

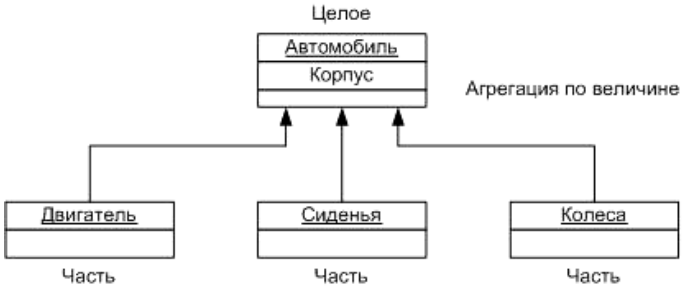

Рис. 2.7. Физическое включение частей в агрегат

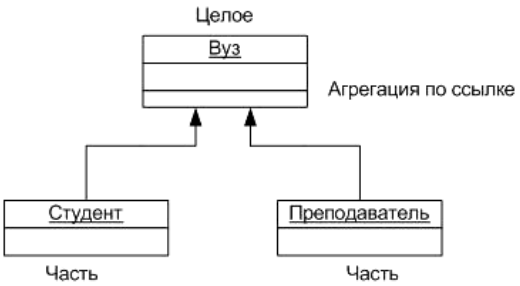

Рис. 2.8. Нефизическое включение частей в агрегат

Итак, между объектами существует два вида отношений – связи и агрегация. Какое из них выбрать?

При выборе вида отношения должны учитываться следующие факторы:

- связи обеспечивают низкое сцепление между объектами;
- агрегация инкапсулирует части как секреты целого.

КЛАССЫ

Понятия объекта и класса тесно связаны. Тем не менее существует важное различие между этими понятиями. Класс – это абстракция существенных характеристик объекта.

ОБЩАЯ ХАРАКТЕРИСТИКА КЛАССОВ

Класс – описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Любой объект – просто экземпляр класса.

Как показано на рисунке 2.9, различают внутреннее представление класса (реализацию) и внешнее представление класса (интерфейс).

Рис. 2.9. Структура представления класса

Интерфейс объявляет возможности (услуги) класса, но скрывает его структуру и поведение. Иными словами, интерфейс демонстрирует внешнему миру абстракцию класса, его внешний облик. Интерфейс в основном состоит из объявлений всех операций, применимых к экземплярам класса. Он может также включать объявления типов, переменных, констант и исключений, необходимых для полноты данной абстракции.

Интерфейс может быть разделён на 3 части:

1) публичную (*public*), объявления которой доступны всем клиентам;

2) защищённую (*protected*), объявления которой доступны только самому классу, его подклассам и друзьям;

3) приватную (*private*), объявления которой доступны только самому классу и его друзьям.

Другом класса называют класс, который имеет доступ ко всем частям этого класса (публичной, защищённой и приватной). Иными словами, от друга у класса нет секретов.

Примечание. Другом класса может быть и свободная подпрограмма.

Реализация класса описывает секреты поведения класса. Она включает реализации всех операций, определённых в интерфейсе класса.

ВИДЫ ОТНОШЕНИЙ МЕЖДУ КЛАССАМИ

Классы, подобно объектам, не существуют в изоляции. Напротив, с отдельной проблемной областью связывают ключевые абстракции, отношения между которыми формируют структуру из классов системы.

Всего существует четыре основных вида отношений между классами:

- ассоциация (фиксирует структурные отношения – связи между экземплярами классов);
- зависимость (отображает влияние одного класса на другой класс);
- обобщение-специализация («is a»-отношение);
- целое-часть («part of»-отношение).

Для покрытия основных отношений большинство объектно-ориентированных языков программирования поддерживает следующие отношения:

1) ассоциация;
2) наследование;
3) агрегация;
4) зависимость;
5) конкретизация;
6) метакласс;
7) реализация.

Ассоциации обеспечивают взаимодействия объектов, принадлежащих разным классам. Они являются клеем, соединяющим воедино все элементы программной системы. Благодаря ассоциациям мы получаем работающую систему. Без ассоциаций система превращается в набор изолированных классов-одиночек.

Наследование – наиболее популярная разновидность отношения *обобщение-специализация*. Альтернативой наследованию считается делегирование. При делегировании объекты *делегируют своё* поведение родственным объектам. При этом классы становятся не нужны.

Агрегация обеспечивает отношения *целое-часть*, объявляемые для экземпляров классов.

Зависимость часто представляется в виде частной формы – *использования*, которое фиксирует отношение между клиентом, запрашивающим услугу, и сервером, предоставляющим эту услугу.

Конкретизация выражает другую разновидность отношения *обобщение-специализация*. Применяется в таких языках, как C++.

Отношения метаклассов поддерживаются в языках SmallTalk и CLOS. Метакласс – это класс классов, понятие, позволяющее обращаться с классами как с объектами.

Реализация определяет отношение, при котором класс-приёмник обеспечивает свою собственную реализацию интерфейса другого класса-источника. Иными словами, здесь идёт речь о наследовании интерфейса.

Семантически реализация – это «скрещивание» отношений зависимости и обобщения-специализации.

АССОЦИАЦИИ КЛАССОВ

Ассоциация обозначает семантическое соединение классов.

П р и м е р: в системе обслуживания читателей имеются две ключевые абстракции – Книга и Библиотека. Класс «Книга» играет роль элемента, хранимого в библиотеке. Класс «Библиотека» играет роль хранилища для книг.

Отношение ассоциации между классами изображено на рис. 2.10. Очевидно, что ассоциация предполагает двухсторонние отношения:

- для данного экземпляра Книги выделяется экземпляр Библиотеки, обеспечивающий её хранение;
- для данного экземпляра Библиотеки выделяются все хранимые Книги.

Здесь показана ассоциация *один-ко-многим*. Каждый экземпляр Книги имеет указатель на экземпляр Библиотеки. Каждый экземпляр Библиотеки имеет набор указателей на несколько экземпляров Книги.

Ассоциация обозначает только семантическую связь. Она не указывает направление и точную реализацию отношения. Ассоциация пригодна для анализа проблемы, когда нам требуется лишь идентифицировать связи. С помощью создания ассоциаций мы приводим к пониманию участников семантических связей, их ролей, мощности (количества элементов).

Ассоциация *один-ко-многим*, введённая в примере, означает, что для каждого экземпляра класса «Библиотека» есть 0 или более экземпляров класса «Книга», а для каждого экземпляра класса «Книга» есть один экземпляр класса «Библиотека». Эту множественность обозначает *мощность ассоциации*. Мощность ассоциации бывает трёх типов:

- один-к-одному;
- один-ко-многим;
- многие-ко-многим.

Примеры ассоциаций с различными типами мощности приведены на рис. 2.11, они имеют следующий смысл:

- у европейской жены один муж, а у европейского мужа одна жена;
- у восточной жены один муж, а у восточного мужа сколько угодно жён;

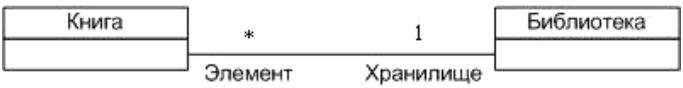

Рис. 2.10. Ассоциация

Рис. 2.11. Ассоциации с различными типами мощности

- у заказа один клиент, а у клиента сколько угодно заказов;
- человек может посещать сколько угодно зданий, а в здании может находиться сколько угодно людей.

НАСЛЕДОВАНИЕ

Наследование – это отношение, при котором один класс разделяет структуру и поведение, определённые в одном другом (простое наследование) или во многих других (множественное наследование) классах.

Между *n* классами наследование определяет иерархию *«является»* (*«is a»*), при которой подкласс наследует от одного или нескольких более общих суперклассов. Говорят, что подкласс *является* специализацией его суперкласса (за счёт дополнения или переопределения существующей структуры или поведения).

Пример: дана система для записи параметров полёта в «чёрный ящик», установленный в самолёте. Организуем систему в виде иерархии классов, построенной на базе наследования. Абстракция «верхнего» класса иерархии имеет вид:

```
with ...;...
use ...; ...
Package Класс «ПараметрыПолета» is
    type «ПараметрыПолета» is tagged private;
    function «Инициировать» return «ПараметрыПолета»;
```

```
    procedure «Записывать» (in out ПараметрыПолета);
    function «ТекущееВремя» (ПараметрыПолета)
        return «БортовоеВремя»;
private
    type «ПараметрыПолета» is tagged record;
        Имя: integer;
    ОтметкаВремени: БортовоеВремя;
    end record;
end Класс «ПараметрыПолета».
```

Запись параметров кабины самолета может обеспечиваться следующим классом:

```
with Класс «ПараметрыПолета»;
use Класс «ПараметрыПолета»;
Package Класс «Кабина» is
    type «Кабина» is new «ПараметрыПолета» with private;
    function Инициировать (Д:Давление; К:Кислород;
        Т:Температура) return «Кабина»;
    procedure «Записывать» (in out Кабина);
    function «ПерепадДавления» (Кабина) return «Давление»;
private
    type «Кабина» is new «ПараметрыПолета»
    with record
        параметр1: Давление;
        параметр2: Кислород;
        параметр3: Температура
    end record;
end Класс «Кабина».
```

Этот класс наследует структуру и поведение класса «ПараметрыПолета», но наращивает его структуру (вводит три новых элемента данных), переопределяет его поведение (процедура «Записывать») и дополняет его поведение (функция «ПерепадДавления»).

Иерархическая структура классов системы для записи параметров полёта, находящихся в отношении наследования, показана на рис. 2.12.

Здесь «ПараметрыПолета» – базовый (корневой) суперкласс, подклассами которого являются Экипаж, ПараметрыДвижения, Приборы, Кабина. В свою очередь, класс «ПараметрыДвижения» является суперклассом для его подклассов «Координаты», «Скорость», «Ориентация».

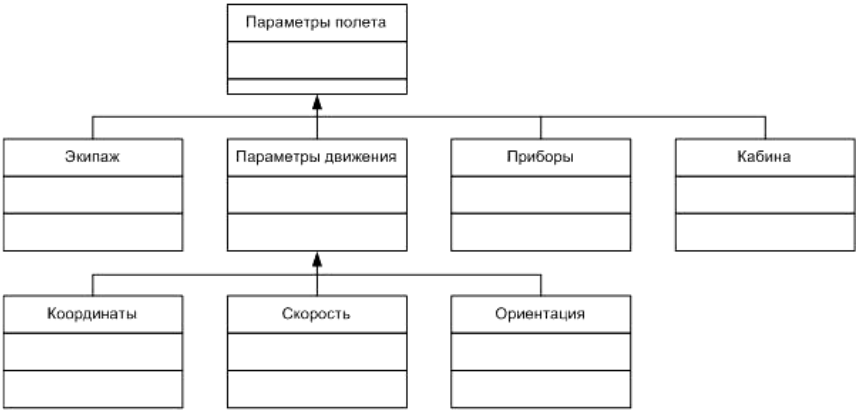

Рис. 2.12. Иерархия простого наследования

ПОЛИМОРФИЗМ

Полиморфизм – возможность с помощью одного имени обозначать операции из различных классов (но относящихся к общему суперклассу). Вызов обслуживания по полиморфному имени приводит к исполнению одной из некоторого набора операций.

Рассмотрим различные реализации процедуры «Записывать». Для класса «ПараметрыПолета» реализация имеет вид

```
procedure «Записывать» (in out ПараметрыПолета) is
begin
    – записывать имя параметра;
    – записывать отметку времени;
end Записывать.
```

В классе «Кабина» предусмотрена другая реализация процедуры

```
procedure Записывать (in out Кабина) is
begin
    Записывать (ПараметрыПолета);
    – вызов метода  суперкласса;
    – записывать значение давления;
    – записывать процентное содержание кислорода;
    – записывать значение температуры;
end Записывать.
```

Предположим, что мы имеем по экземпляру каждого из этих двух классов:

```
В полете: ПараметрыПолета:= Инициировать;
В кабине: Кабина:= Инициировать (768. 21. 20);
Предположим также, что имеется свободная процедура:
procedure СохранятьНовыеДанные (d: in out
        ПараметрыПолета'class; t: БортовоеВремя) is
begin
    if ТекущееВремя(d) >= t then
        Записывать (d): – диспетчирование с помощью тега
    end if;
end СохранятьНовыеДанные.
```

Что случится при выполнении следующих операторов?

- СохранятьНовыеДанные (Вполете, БортовоеВремя (60));
- СохранятьНовыеДанные (Вкабине, БортовоеВремя (120)).

Каждый из операторов вызывает операцию «Записывать» нужного класса. В первом случае диспетчеризация приведёт к операции «Записывать» из класса «ПараметрыПолета». Во втором случае будет выполняться операция из класса «Кабина». Как видим, в свободной процедуре переменная d может обозначать объекты разных классов, значит, здесь записан вызов полиморфной операции.

АГРЕГАЦИЯ

Отношения агрегации между классами аналогичны отношениям агрегации между объектами.

Повторим пример с описанием класса «КонтроллерУгла»:

```
with Класс «ГрафикРазворота». Класс «РегуляторУгла»;
use Класс «ГрафикРазворота», Класс «РегуляторУгла»;
Package Класс «КонтроллерУгла» is
    type указатель наГрафик is access all ГрафикРазворота;
    type КонтроллерУгла is tagged private:
    procedure Обрабатывать (in out КонтроллерУгла;
        угол: указатель на График);
function Запланировано (КонтроллерУгла;
        угол: указатель на График) return Секунда;
private
    type КонтроллерУгла is tagged record;
    регулятор: РегуляторУгла;
    …
end Класс «КонтроллерУгла».
```

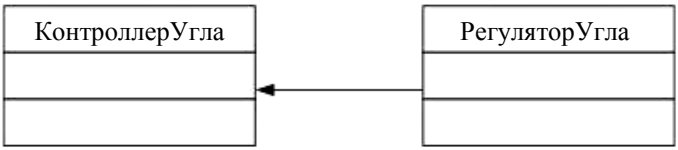

Рис. 2.13. Отношение агрегации по величине (композиция)

Видим, что класс «КонтроллерУгла» является агрегатом, а экземпляр класса «РегуляторУгла» – это одна из его частей. Агрегация здесь определена как включение по величине. Это пример физического включения, означающий, что объект «Регулятор» не существует независимо от включающего его экземпляра КонтроллераУгла. Время жизни этих двух объектов неразрывно связано.

Графическая иллюстрация отношения агрегации по величине (композиции) представлена на рис. 2.13.

Возможен косвенный тип агрегации – включение по ссылке. Если мы запишем в приватной части класса «КонтроллерУгла»:

```
…
private
    type указатель на РегуляторУгла is access all РегуляторУгла;
    type КонтроллерУгла is tagged record;
        регулятор: указатель на РегуляторУгла;
    …
end Класс «КонтроллерУгла»;
```

то регулятор как часть контроллера будет доступен косвенно.

Теперь сцепление объектов уменьшено. Экземпляры каждого класса создаются и уничтожаются независимо.

Ещё два примера агрегации по ссылке и по величине (композиции) приведены на рис. 2.14. Здесь показаны класс-агрегат «Дом» и класс-агрегат «Окно», причём указаны роли и множественность частей агрегата (соответствующие пометки имеют линии отношений).

Как показано на рисунке 2.15, возможны и другие формы представления агрегации по величине – композиции. Композицию можно отобразить графическим вложением символов частей в символ агрегата (левая часть рис. 2.15). Вложенные части демонстрируют свою множественность (мощность, кратность) в правом верхнем углу своего символа. Если метка множественности опущена, по умолчанию считают, что её значение «много». Вложенный элемент может иметь роль в агрегате. Используется синтаксис

```
роль : имя Класса.
```

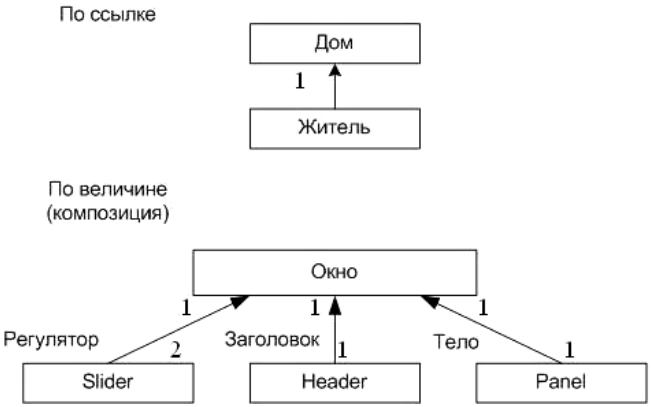

Рис. 2.14. Агрегация классов

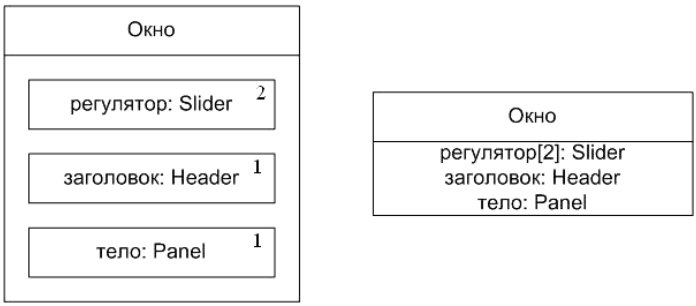

Рис. 2.15. Формы представления композиции

Эта роль соответствует той роли, которую играет часть в неявном (в этой нотации) отношении композиции между частью и целым (агрегатом).

Как представлено в правой части рис. 2.15, свойства (атрибуты) класса находятся в отношении композиции между всем классом и его элементами-свойствами. Тем не менее в общем случае свойства должны иметь примитивные значения (числа, строки, даты), а не ссылаться на другие классы, так как в «атрибутной» нотации не видны другие отношения классов-частей. Кроме того, свойства классов не могут находиться в совместном использовании несколькими классами.

ЗАВИСИМОСТЬ

Зависимость – это отношение, которое показывает, что изменение в одном классе (независимом) может влиять на другой класс (зависимый), который использует его. Графически зависимость изображается как пунк-

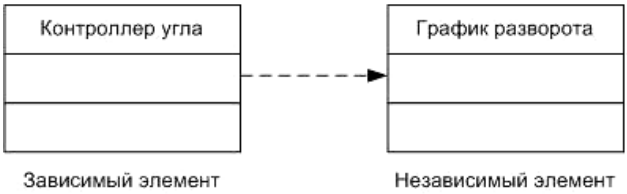

Рис. 2.16. Отношение зависимости

тирная стрелка, направленная на класс, от которого зависят. С помощью зависимости уточняют, какая абстракция является клиентом, а какая – поставщиком определённой услуги. Пунктирная стрелка зависимости направлена от клиента к поставщику.

Наиболее часто зависимости показывают, что один класс использует другой класс как аргумент в сигнатуре своей операции. В предыдущем примере класс «ГрафикРазворота» появляется как аргумент в методах «Обрабатывать» и «Запланировано» класса «КонтроллерУгла». Поэтому, как показано на рис. 2.16, КонтроллерУгла зависит от класса «ГрафикРазворота».

КОНКРЕТИЗАЦИЯ

В [11] определяется конкретизация как процесс наполнения шаблона (родового или параметризованного класса). Целью является получение класса, от которого возможно создание экземпляров.

Родовой класс служит заготовкой, шаблоном, параметры которого могут наполняться (настраиваться) другими классами, типами, объектами, операциями. Он может быть родоначальником большого количества обычных (конкретных) классов. Возможности настройки родового класса представляются списком формальных родовых параметров. Эти параметры в процессе настройки должны заменяться фактическими родовыми параметрами. Процесс настройки родового класса называют конкретизацией.

В разных языках программирования родовые классы оформляются по-разному. Воспользуемся возможностями языка, в котором впервые была реализована идея настройки-параметризации. Здесь формальные родовые параметры записываются между словом *generic* и заголовком пакета, размещающего класс.

П р и м е р: представим родовой (параметризированный) класс «Очередь»:

```
generic
    type Элемент is private;
```

```
package Класс «Очередь» is
    type Очередь is limited tagged private;
    …
    procedure Добавить (В Очередь: in out Очередь;
        элемент: Элемент);
    …
private
    …
end Класс «Очередь».
```

У этого класса один формальный родовой параметр – тип «Элемент». Вместо этого параметра можно подставить почти любой тип данных.

Произведём настройку, т.е. объявим два конкретизированных класса – «ОчередьЦелыхЭлементов» и «ОчередьЛилипутов»:

```
package Класс «ОчередьЦелыхЭлементов» is new Класс «Очередь»
        (Элемент => Integer);
package Класс «ОчередьЛилипутов» is new Класс «Очередь»
        (Элемент => Лилипут).
```

В первом случае мы настраивали класс на конкретный тип «Integer» (фактический родовой параметр), во втором случае – на конкретный тип «Лилипут».

Классы «ОчередьЦелыхЭлементов» и «ОчередьЛилипутов» можно использовать как обычные классы. Они содержат все средства родового класса, но только эти средства настроены на использование конкретного типа, заданного при конкретизации.

Графическая иллюстрация отношений конкретизации приведена на рис. 2.17. Отметим, что отношение конкретизации отображается с помощью подписанной стрелки отношения зависимости. Это логично, поскольку конкретизированный класс зависит от родового класса (класса-шаблона).

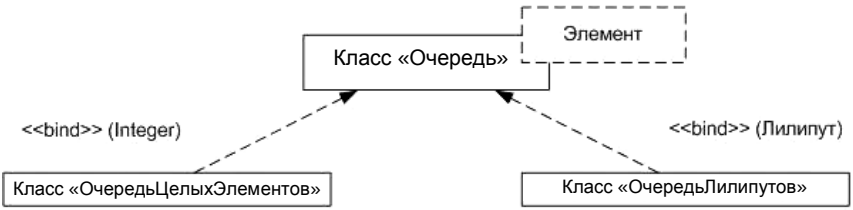

Рис. 2.17. Отношения конкретизации родового класса

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чём отличие алгоритмической декомпозиции от объектно-ориентированной декомпозиции сложной системы?
2. В чём особенность объектно-ориентированного абстрагирования?
3. В чём особенность объектно-ориентированной инкапсуляции?
4. Каковы средства обеспечения объектно-ориентированной модульности?
5. Каковы особенности объектно-ориентированной иерархии? Какие разновидности этой иерархии вы знаете?
6. Дайте общую характеристику объектов.
7. Что такое состояние объекта?
8. Что такое поведение объекта?
9. Какие виды операций вы знаете?
10. Что такое протокол объекта?
11. Что такое обязанности объекта?
12. Чем отличаются активные объекты от пассивных объектов?
13. Что такое роли объектов?
14. Чем отличается объект от класса?
15. Охарактеризуйте связи между объектами.
16. Охарактеризуйте роли объектов в связях.
17. Какие формы видимости между объектами вы знаете?
18. Охарактеризуйте отношение агрегации между объектами. Какие разновидности агрегации вы знаете?
19. Дайте общую характеристику класса.
20. Поясните внутреннее и внешнее представление класса.
21. Какие вы знаете секции в интерфейсной части класса?

3. БАЗИС ЯЗЫКА ВИЗУАЛЬНОГО МОДЕЛИРОВАНИЯ

Для создания моделей анализа и проектирования объектно-ориентированных программных систем используют языки визуального моделирования. Появившись сравнительно недавно, в период с 1989 по 1997 гг., эти языки уже имеют представительную историю развития.

В настоящее время различают три поколения языков визуального моделирования. И если первое поколение образовали 10 языков, то численность второго поколения уже превысила 50 языков. Среди наиболее популярных языков второго поколения можно выделить: язык Буча (G. Booch), язык Рамбо (J. Rumbaugh), язык Джекобсона (I. Jacobson), язык Коада-Йордона (Coad-Yourdon), язык Шлеера-Меллора (Shlaer-Mellor) и т.д. [41], [64], [69]. Каждый язык вводил свои выразительные средства, ориентировался на собственный синтаксис и семантику, иными словами – претендовал на роль единственного и неповторимого языка. В результате разработчики (и пользователи этих языков) перестали понимать друг друга. Возникла острая необходимость унификации языков.

Идея унификации привела к появлению языков третьего поколения. В качестве стандартного языка третьего поколения был принят Unified Modeling Language (UML), создававшийся в 1994 – 1997 гг. (основные разработчики – три «amigos» Г. Буч, Дж. Рамбо, И. Джекобсон). В настоящее время разработана версия UML 2.0, которая описывается в [17]. Данная глава посвящена определению базовых понятий языка UML.

УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ

UML – стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем [17], [18], [19]. UML может использоваться для визуализации, спецификации, конструирования и документирования результатов программных проектов. UML – это не визуальный язык программирования, однако его модели прямо транслируются в текст на языках программирования (Java, C++, Visual Basic, Object Pascal) и даже в таблицы для реляционной БД.

Словарь UML образуют три разновидности строительных блоков: предметы, отношения, диаграммы.

Предметы – это абстракции, которые являются основными элементами в модели, отношения связывают эти предметы, диаграммы группируют коллекции предметов.

ПРЕДМЕТЫ В UML

В UML имеются четыре разновидности предметов:

- структурные предметы;
- предметы поведения;
- группирующие предметы;
- поясняющие предметы.

Эти предметы являются базовыми объектно-ориентированными строительными блоками. Они используются для написания моделей.

Структурные предметы являются существительными в UML-моделях. Они представляют статические части модели – понятийные или физические элементы. Перечислим восемь разновидностей структурных предметов.

Рис. 3.1. Классы

1. *Класс* – описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Класс реализует один или несколько интерфейсов. Как показано на рис. 3.1, графически класс отображается в виде прямоугольника, обычно включающего секции с именем, свойствами (атрибутами) и операциями.

Рис. 3.2. Интерфейсы

2. *Интерфейс* – набор операций, которые определяют услуги класса или компонента. Интерфейс описывает поведение элемента, видимое извне. Интерфейс может представлять полные услуги класса или компонента или часть таких услуг. Интерфейс определяет набор спецификаций операций (их сигнатуры), а не набор реализаций операций. Графически интерфейс изображается в виде кружка с именем, как показано на рис. 3.2. Имя интерфейса обычно начинается с буквы «I». Интерфейс редко показывают самостоятельно. Обычно его присоединяют к классу или компоненту, который реализует интерфейс.

3. *Кооперация* (сотрудничество) определяет взаимодействие и является совокупностью ролей и других элементов, которые работают вместе для обеспечения коллективного поведения более сложного, чем простая сумма всех элементов. Таким образом, кооперации имеют как структурное, так и поведенческое измерения. Конкретный класс может участвовать в нескольких кооперациях. Эти кооперации представляют реализацию паттернов (образцов), которые формируют систему. Как показано на рис. 3.3, графически кооперация изображается как пунктирный эллипс, в который вписывается её имя.

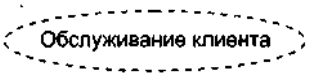

Рис. 3.3. Кооперации

4. *Актёр* – набор согласованных ролей, которые могут играть пользователи при взаимодействии с системой (её элементами Use Case). Каждая роль требует от системы определённого поведения. Как показано на рис. 3.4, актёр изображается как проволочный человечек с именем.

Рис. 3.4. Актёры

5. *Элемент Use Case* (Прецедент) – описание последовательности действий (или нескольких последовательностей), выполняемых системой в интересах отдельного актёра и производящих видимый для актёра результат. В модели элемент Use Case применяется для структурирования предметов поведения. Элемент Use Case реализуется кооперацией. Как показано на рис. 3.5, элемент Use Case изображается как эллипс, в который вписывается его имя.

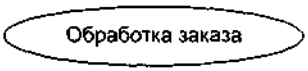

Рис. 3.5. Элементы Use Case

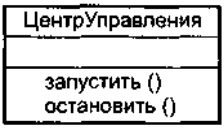

Рис. 3.6. Активные классы

6. *Активный класс* – класс, чьи объекты имеют один или несколько процессов (или потоков) и поэтому могут инициировать управляющую деятельность. Активный класс похож на обычный класс за исключением того, что его объекты действуют одновременно с объектами других классов. Как показано на рис. 3.6, активный класс изображается как утолщённый прямоугольник, обычно включающий имя, свойства (атрибуты) и операции.

7. *Компонент* – физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов. В систему включаются как компоненты, являющиеся результатами процесса разработки (файлы исходного кода), так и различные разновидности используемых компонентов (COM+-компоненты, Java Beans). Обычно, компонент – это физическая упаковка различных логических элементов (классов, интерфейсов и сотрудничеств). Как показано на рис. 3.7, компонент изображается как прямоугольник с вкладками, обычно включающий имя.

Рис. 3.7. Компоненты

8. *Узел* – физический элемент, который существует в период работы системы и представляет ресурс, обычно имеющий память и возможности обработки. В узле размещается набор компонентов, который может перемещаться от узла к узлу. Как показано на рис. 3.8, узел изображается как куб с именем.

Рис. 3.8. Узлы

Предметы поведения – динамические части UML-моделей. Они являются глаголами моделей, представлением поведения во времени и пространстве. Существует две основные разновидности предметов поведения.

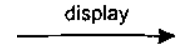

Рис. 3.9. Сообщения

1. *Взаимодействие* – поведение, заключающее в себе набор сообщений, которыми обменивается набор объектов в конкретном контексте для достижения определённой цели. Взаимодействие может определять динамику как совокупности объектов, так и отдельной операции. Элементами взаимодействия являются сообщения, последовательность действий (поведение, вызываемое сообщением) и связи (соединения между объектами). Как показано на рис. 3.9, сообщение изображается в виде направленной линии с именем её операции.

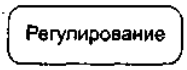

Рис. 3.10. Состояния

2. *Конечный автомат* – поведение, которое определяет последовательность состояний объекта или взаимодействия, выполняемые в ходе его существования в ответ на события (и с учётом обязанностей по этим событиям). С помощью конечного автомата может определяться поведение индивидуального класса или кооперации классов. Элементами конечного автомата являются состояния, переходы (от состояния к состоянию), события (предметы, вызывающие переходы) и действия (реакции на переход). Как показано на рис. 3.10, состояние изображается как закруглённый прямоугольник, обычно включающий его имя и его подсостояния (если они есть).

Эти два элемента – взаимодействия и конечные автоматы – являются базисными предметами поведения, которые могут включаться в UML-модели. Семантически эти элементы ассоциируются с различными структурными элементами (прежде всего с классами, сотрудничествами и объектами).

Группирующие предметы – организационные части UML-моделей. Это ящики, по которым может быть разложена модель. Предусмотрена одна разновидность группирующего предмета – пакет.

Рис. 3.11. Пакеты

Пакет – общий механизм для распределения элементов по группам. В пакет могут помещаться структурные предметы, предметы поведения и даже другие группировки предметов. В отличие от компонента (который существует в период выполнения), пакет – чисто концептуальное понятие. Это означает, что пакет существует только в период разработки. Как показано на рис. 3.11, пакет изображается как папка с закладкой, на которой обозначено его имя и, иногда, его содержание.

Поясняющие предметы – разъясняющие части UML-моделей. Они являются замечаниями, которые можно применить для описания, объяснения и комментирования любого элемента модели. Предусмотрена одна разновидность поясняющего предмета – примечание.

Примечание – символ для отображения ограничений и замечаний, присоединяемых к элементу или совокупности элементов. Как показано на рис. 3.12, примечание изображается в виде прямоугольника с загнутым углом, в который вписывается текстовый или графический комментарий.

Рис. 3.12. Примечания

ОТНОШЕНИЯ В UML

В UML имеются четыре разновидности отношений:

1) зависимость;
2) ассоциация;
3) обобщение;
4) реализация.

Эти отношения являются базовыми строительными блоками отношений. Они используются при написании моделей.

1. *Зависимость* – семантическое отношение между двумя предметами, в котором изменение в одном предмете (независимом предмете) может влиять на семантику другого предмета (зависимого предмета). Как показано на рис. 3.13, зависимость изображается в виде пунктирной линии, возможно направленной на независимый предмет и иногда имеющей метку.

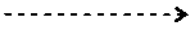

Рис. 3.13. Зависимости

2. *Ассоциация* – структурное отношение, которое описывает набор связей, являющихся соединением между объектами. Агрегация – это специальная разновидность ассоциации, представляющая структурное отношение между целым и его частями. Как показано на рис. 3.14, ассоциация изображается в виде сплошной линии, возможно направленной, иногда имеющей метку и часто включающей другие «украшения», такие как мощность и имена ролей.

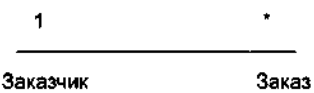

Рис. 3.14. Ассоциации

3. *Обобщение* – отношение специализации/обобщения, в котором объекты специализированного элемента (потомка, ребенка) могут заменять объекты обобщённого элемента (предка, родителя). Иначе говоря, потомок разделяет структуру и поведение родителя. Как показано на рис. 3.15, обобщение изображается в виде сплошной стрелки с полым наконечником, указывающим на родителя.

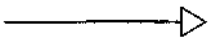

Рис. 3.15. Обобщения

4. *Реализация* – семантическое отношение между классификаторами, где один классификатор определяет контракт, который другой классификатор обязуется выполнять (к классификаторам относят классы, интерфейсы, компоненты, элементы Use Case, кооперации). Отношения

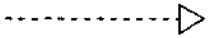

Рис. 3.16. Реализации

реализации применяют в двух случаях: между интерфейсами и классами (или компонентами), реализующими их; между элементами Use Case и кооперациями, которые реализуют их. Как показано на рис. 3.16, реализация изображается как нечто среднее между обобщением и зависимостью.

ДИАГРАММЫ В UML

Диаграмма – графическое представление множества элементов, наиболее часто изображается как связный граф из вершин (предметов) и дуг (отношений). Диаграммы рисуются для визуализации системы с разных точек зрения, затем они отображаются в систему. Обычно диаграмма даёт неполное представление элементов, которые составляют систему. Хотя один и тот же элемент может появляться во всех диаграммах, на практике он появляется только в некоторых диаграммах. Теоретически диаграмма может содержать любую комбинацию предметов и отношений, на практике ограничиваются малым количеством комбинаций, которые соответствуют пяти представлениям архитектуры ПС. По этой причине UML включает девять видов диаграмм:

1) диаграммы классов;
2) диаграммы объектов;
3) диаграммы Use Case (диаграммы прецедентов);
4) диаграммы последовательности;
5) диаграммы сотрудничества (кооперации);
6) диаграммы схем состояний;
7) диаграммы деятельности;
8) компонентные диаграммы;
9) диаграммы размещения (развёртывания).

Диаграмма классов показывает набор классов, интерфейсов, сотрудничеств и их отношений. При моделировании объектно-ориентированных систем диаграммы классов используются наиболее часто. Диаграммы классов обеспечивают статическое проектное представление системы. Диаграммы классов, включающие активные классы, обеспечивают статическое представление процессов системы.

Диаграмма объектов показывает набор объектов и их отношения. Диаграмма объектов представляет статический «моментальный снимок» с экземпляров предметов, которые находятся в диаграммах классов. Как и диаграммы классов, эти диаграммы обеспечивают статическое проектное представление или статическое представление процессов системы (но с точки зрения реальных или фототипичных случаев).

Диаграмма Use Case (диаграмма прецедентов) показывает набор элементов Use Case, актёров и их отношений. С помощью диаграмм Use Case для системы создаётся статическое представление Use Case. Эти диа-

граммы особенно важны при организации и моделировании поведения системы, задании требований заказчика к системе.

Диаграммы последовательности и диаграммы сотрудничества – это разновидности диаграмм взаимодействия.

Диаграмма взаимодействия показывает взаимодействие, включающее набор объектов и их отношений, а также пересылаемые между объектами сообщения. Диаграммы взаимодействия обеспечивают динамическое представление системы.

Диаграмма последовательности – это диаграмма взаимодействия, которая выделяет упорядочение сообщений по времени.

Диаграмма сотрудничества (диаграмма кооперации) – это диаграмма взаимодействия, которая выделяет структурную организацию объектов, посылающих и принимающих сообщения. Диаграммы последовательности и диаграммы сотрудничества изоморфны, что означает, что одну диаграмму можно трансформировать в другую диаграмму.

Диаграмма схем состояний показывает конечный автомат, представляет состояния, переходы, события и действия. Диаграммы схем состояний обеспечивают динамическое представление системы. Они особенно важны при моделировании поведения интерфейса, класса или сотрудничества. Эти диаграммы выделяют такое поведение объекта, которое управляется событиями, что особенно полезно при моделировании реактивных систем.

Диаграмма деятельности – специальная разновидность диаграммы схем состояний, которая показывает поток от действия к действию внутри системы. Диаграммы деятельности обеспечивают динамическое представление системы. Они особенно важны при моделировании функциональности системы и выделяют поток управления между объектами.

Компонентная диаграмма показывает организацию набора компонентов и зависимости между компонентами. Компонентные диаграммы обеспечивают статическое представление реализации системы. Они связаны с диаграммами классов в том смысле, что в компонент обычно отображается один или несколько классов, интерфейсов или кооперации.

Диаграмма размещения (диаграмма развёртывания) показывает конфигурацию обрабатывающих узлов периода выполнения, а также компоненты, живущие в них. Диаграммы размещения обеспечивают статическое представление размещения системы. Они связаны с компонентными диаграммами в том смысле, что узел обычно включает один или несколько компонентов.

МЕХАНИЗМЫ РАСШИРЕНИЯ В UML

UML – развитый язык, имеющий большие возможности, но даже он не может отразить все нюансы, которые могут возникнуть при создании различных моделей. Поэтому UML создавался как открытый язык, допус-

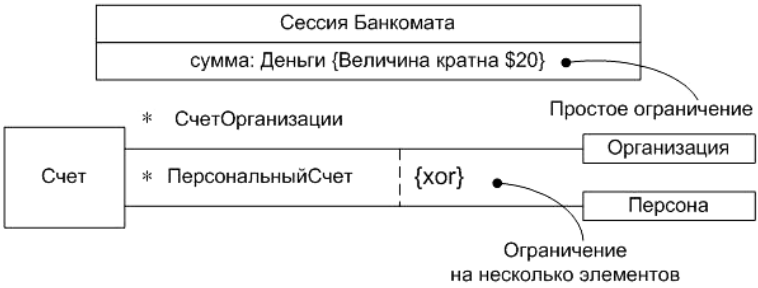

Рис. 3.17. Ограничения

кающий контролируемые расширения. Механизмами расширения в UML являются:

- ограничения;
- теговые величины;
- стереотипы.

Ограничение (constraint) расширяет семантику строительного UML-блока, позволяя добавить новые правила или модифицировать существующие. Ограничение показывают как текстовую строку, заключённую в фигурные скобки {}. Например, на рисунке 3.17 введено простое ограничение на свойство *Сумма* класса *СессияБанкомата* – его значение должно быть кратно 20. Кроме того, здесь показано ограничение на два элемента (две ассоциации), оно располагается возле пунктирной линии, соединяющей элементы, и имеет следующий смысл – владельцем конкретного счёта не может быть и организация, и персона.

Теговая величина (tagged value) расширяет характеристики строительного UML-блока, позволяя создать новую информацию в спецификации конкретного элемента. Теговую величину показывают как строку в фигурных скобках {}. Строка имеет вид

имя теговой величины = значение.

Иногда (в случае предопределённых тегов) указывается только имя теговой величины.

Отметим, что при работе с продуктом, имеющим много реализаций, полезно отслеживать версию и автора определённых блоков. Версия и автор не принадлежат к основным понятиям UML. Они могут быть добавлены к любому строительному блоку (например, к классу) введением в блок новых теговых величин. Например, на рисунке 3.18 класс «ТекстовыйПроцессор» расширен путём явного указания его версии и автора.

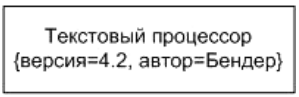

Рис. 3.18. Расширение класса

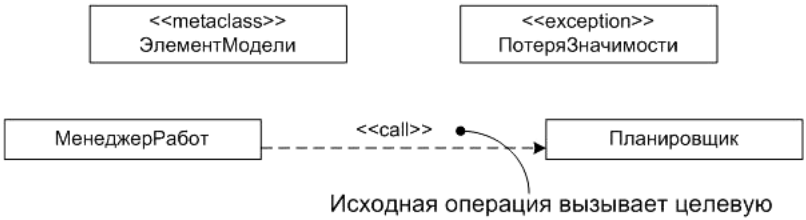

Рис. 3.19. Стереотипы

Стереотип (stereotype) расширяет словарь языка, позволяет создавать новые виды строительных блоков, производные от существующих и учитывающие специфику новой проблемы. Элемент со стереотипом является вариацией существующего элемента, имеющей такую же форму, но отличающуюся по сути. У него могут быть дополнительные ограничения и теговые величины, а также другое визуальное представление. Он иначе обрабатывается при генерации программного кода. Отображают стереотип как имя, указываемое в двойных угловых скобках (или в угловых кавычках).

Примеры элементов со стереотипами приведены на рис. 3.19. Стереотип «exception» говорит о том, что класс «ПотеряЗначимости» теперь рассматривается как специальный класс, которому, положим, разрешается только генерация и обработка сигналов-исключений. Особые возможности метакласса получил класс «ЭлементМодели». Кроме того, здесь показано применение стереотипа «call» к отношению зависимости (у него появился новый смысл).

Таким образом, механизмы расширения позволяют адаптировать UML под нужды конкретных проектов и под новые программные технологии. Возможно добавление новых строительных блоков, модификация спецификаций существующих блоков и даже изменение их семантики. Конечно, очень важно обеспечить контролируемое введение расширений.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Сколько поколений языков визуального моделирования вы знаете?
2. Назовите численность языков визуального моделирования второго поколения.
3. Какая необходимость привела к созданию языка визуального моделирования третьего поколения?
4. Поясните назначение UML.
5. Какие строительные блоки образуют словарь UML? Охарактеризуйте их.
6. Какие разновидности предметов UML вы знаете? Их назначение?

7. Перечислите известные вам разновидности структурных предметов UML.

8. Перечислите известные вам разновидности предметов поведения UML.

9. Перечислите известные вам группирующие предметы UML.

10. Перечислите известные вам поясняющие предметы UML.

11. Какие разновидности отношений предусмотрены в UML? Охарактеризуйте каждое отношение.

12. Дайте характеристику диаграммы классов.

13. Дайте характеристику диаграммы объектов.

14. Охарактеризуйте диаграмму Use Case.

15. Охарактеризуйте диаграммы взаимодействия.

16. Дайте характеристику диаграммы последовательности.

17. Дайте характеристику диаграммы сотрудничества.

18. Охарактеризуйте диаграмму схем состояний.

19. Охарактеризуйте диаграмму деятельности.

20. Дайте характеристику компонентной диаграммы.

21. Охарактеризуйте диаграмму размещения.

22. Для чего служат механизмы расширения в UML?

23. Поясните механизм ограничений в UML.

24. Объясните механизм теговых величин в UML.

25. В чём суть механизма стереотипов UML?

4. ОРГАНИЗАЦИЯ ПРОЦЕССА КОНСТРУИРОВАНИЯ

Основными составляющими технологии конструирования ПО являются продукты (программные системы) и процессы, обеспечивающие создание продуктов. Здесь рассматриваются основные подходы к организации процесса конструирования. Приводятся примеры классических, современных и перспективных процессов конструирования, обсуждаются модели качества процессов конструирования.

ОПРЕДЕЛЕНИЕ ТЕХНОЛОГИИ КОНСТРУИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Технология конструирования программного обеспечения (ТКПО) – система инженерных принципов для создания экономичного ПО, которое надёжно и эффективно работает в реальных компьютерах [15], [20].

Различают методы, средства и процедуры ТКПО.

Методы обеспечивают решение следующих задач:

- планирование и оценка проекта;
- анализ системных и программных требований;
- проектирование алгоритмов, структур данных и программных структур;
- кодирование;
- тестирование;
- сопровождение.

Средства (утилиты) ТКПО обеспечивают автоматизированную или автоматическую поддержку методов. В целях совместного применения утилиты могут объединяться в системы автоматизированного конструирования ПО. Такие системы принято называть CASE-системами. Аббревиатура CASE расшифровывается как Computer Aided Software Engineering (программная инженерия с компьютерной поддержкой).

Процедуры являются «клеем», который соединяет методы и утилиты так, что они обеспечивают непрерывную технологическую цепочку разработки. Процедуры определяют:

- порядок применения методов и утилит;
- формирование отчётов, форм по соответствующим требованиям;
- контроль, который помогает обеспечивать качество и координировать изменения;
- формирование «вех», по которым руководители оценивают прогресс.

Процесс конструирования программного обеспечения состоит из последовательности шагов, использующих методы, утилиты и процедуры. Эти последовательности шагов часто называют парадигмами ТКПО.

Применение парадигм ТКПО гарантирует систематический, упорядоченный подход к промышленной разработке, использованию и сопровождению ПО. Фактически, парадигмы вносят в процесс создания ПО организующее инженерное начало, необходимость которого трудно переоценить.

Рассмотрим наиболее популярные парадигмы ТКПО.

КЛАССИЧЕСКИЙ ЖИЗНЕННЫЙ ЦИКЛ

Старейшей парадигмой процесса разработки ПО является классический жизненный цикл (автор Уинстон Ройс, 1970) [21].

Очень часто классический жизненный цикл называют каскадной или водопадной моделью, подчёркивая, что разработка рассматривается как последовательность этапов, причём переход на следующий, иерархически нижний этап происходит только после полного завершения работ на текущем этапе (рис. 4.1).

Охарактеризуем содержание основных этапов.

Подразумевается, что разработка начинается на системном уровне и проходит через анализ, проектирование, кодирование, тестирование и сопровождение. При этом моделируются действия стандартного инженерного цикла.

Системный анализ задаёт роль каждого элемента в компьютерной системе, взаимодействие элементов друг с другом. Поскольку ПО является лишь частью большой системы, то анализ начинается с определения требований ко всем системным элементам и назначения подмножества

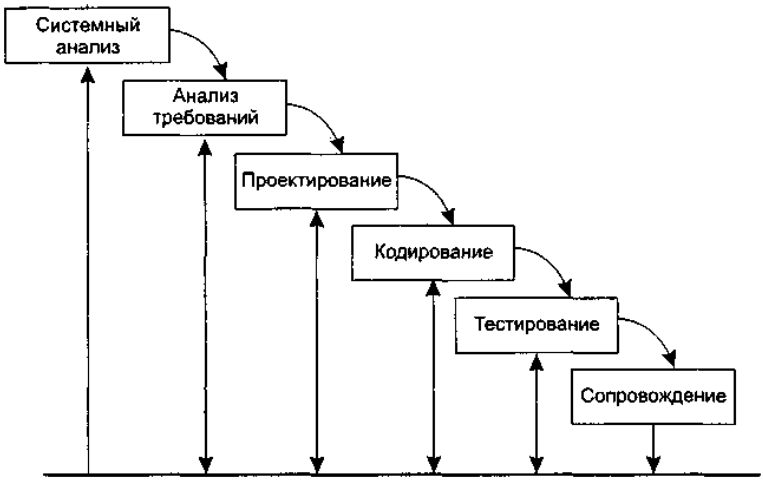

Рис. 4.1. Классический жизненный цикл разработки ПО

этих требований программному «элементу». Необходимость системного подхода явно проявляется, когда формируется интерфейс ПО с другими элементами (аппаратурой, людьми, базами данных). На этом же этапе начинается решение задачи планирования проекта ПО. В ходе планирования проекта определяются объём проектных работ и их риск, необходимые трудозатраты, формируются рабочие задачи и план-график работ.

Анализ требований относится к программному элементу – программному обеспечению. Уточняются и детализируются его функции, характеристики и интерфейс.

Все определения документируются в *спецификации анализа*. Здесь же завершается решение задачи планирования проекта.

Проектирование состоит в создании представлений:

- архитектуры ПО;
- модульной структуры ПО;
- алгоритмической структуры ПО;
- структуры данных;
- входного и выходного интерфейса (входных и выходных форм данных).

Исходные данные для проектирования содержатся в *спецификации анализа*, т.е. в ходе проектирования выполняется трансляция требований к ПО во множество проектных представлений. При решении задач проектирования основное внимание уделяется качеству будущего программного продукта.

Кодирование состоит в переводе результатов проектирования в текст на языке программирования.

Тестирование – выполнение программы для выявления дефектов в функциях, логике и форме реализации программного продукта.

Сопровождение – это внесение изменений в эксплуатируемое ПО. Цели изменений:

- исправление ошибок;
- адаптация к изменениям внешней для ПО среды;
- усовершенствование ПО по требованиям заказчика.

Сопровождение ПО состоит в повторном применении каждого из предшествующих шагов (этапов) жизненного цикла к существующей программе, но не в разработке новой программы.

Как и любая инженерная схема, классический жизненный цикл имеет достоинства и недостатки.

Достоинства классического жизненного цикла: даёт план и временной график по всем этапам проекта, упорядочивает ход конструирования.

Недостатки классического жизненного цикла:

1) реальные проекты часто требуют отклонения от стандартной последовательности шагов;

2) цикл основан на точной формулировке исходных требований к ПО (реально в начале проекта требования заказчика определены лишь частично);

3) результаты проекта доступны заказчику только в конце работы.

МАКЕТИРОВАНИЕ

Достаточно часто заказчик не может сформулировать подробные требования по вводу, обработке или выводу данных для будущего программного продукта. С другой стороны, разработчик может сомневаться в приспосабливаемых свойствах продукта под операционную систему, форме диалога с пользователем или в эффективности реализуемого алгоритма. В этих случаях целесообразно использовать макетирование.

Основная цель макетирования – снять неопределённости в требованиях заказчика.

Макетирование (прототипирование) – это процесс создания модели требуемого программного продукта.

Модель может принимать одну из трёх форм:

1) бумажный макет или макет на основе ПК (изображает или рисует человеко-машинный диалог);

2) работающий макет (выполняет некоторую часть требуемых функций);

3) существующая программа (характеристики которой затем должны быть улучшены).

Как показано на рис. 4.2, макетирование основывается на многократном повторении итераций, в которых участвуют заказчик и разработчик.

Последовательность действий при макетировании представлена на рис. 4.3. Макетирование начинается со сбора и уточнения требований к создаваемому ПО. Разработчик и заказчик встречаются и определяют все цели ПО, устанавливают, какие требования известны, а какие предстоит доопределить.

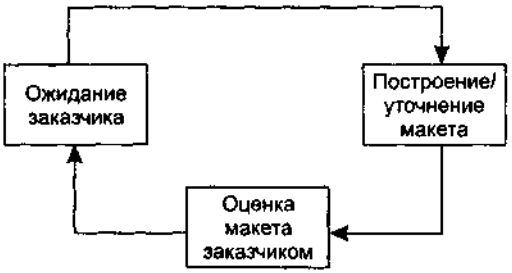

Рис. 4.2. Макетирование

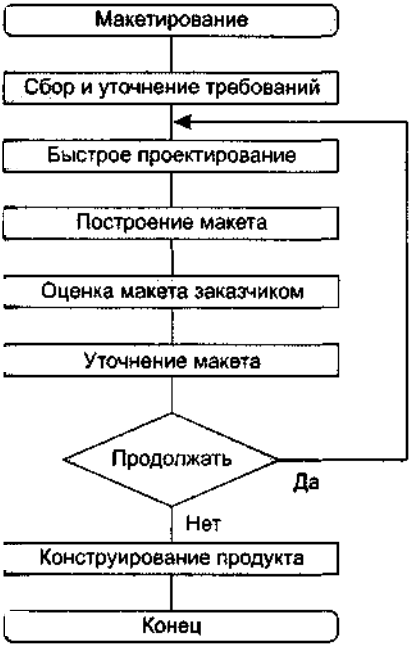

Рис. 4.3. Последовательность действий при макетировании

Затем выполняется быстрое проектирование. В нём внимание сосредоточивается на тех характеристиках ПО, которые должны быть видимы пользователю.

Быстрое проектирование приводит к построению макета.

Макет оценивается заказчиком и используется для уточнения требований к ПО.

Итерации повторяются до тех пор, пока макет не выявит все требования заказчика и, тем самым, не даст возможность разработчику понять, что должно быть сделано.

Достоинство макетирования: обеспечивает определение полных требований к ПО.

Недостатки макетирования:

- заказчик может принять макет за продукт;
- разработчик может принять макет за продукт.

Поясним суть недостатков. Когда заказчик видит работающую версию ПО, он перестаёт сознавать, что детали макета скреплены «жевательной резинкой и проволокой»; он забывает, что в погоне за работающим вариантом оставлены нерешённые вопросы качества и удобства сопровождения ПО. Когда заказчику говорят, что продукт должен быть перестро-

ен, он начинает возмущаться и требовать, чтобы макет «в три приёма» был превращён в рабочий продукт. Очень часто это отрицательно сказывается на управлении разработкой ПО.

С другой стороны, для быстрого получения работающего макета разработчик часто идёт на определённые компромиссы. Могут использоваться не самые подходящие язык программирования или операционная система. Для простой демонстрации возможностей может применяться неэффективный алгоритм. Спустя некоторое время разработчик забывает о причинах, по которым эти средства не подходят. В результате, далеко не идеальный выбранный вариант интегрируется в систему.

Очевидно, что преодоление этих недостатков требует борьбы с житейским соблазном – принять желаемое за действительное.

СТРАТЕГИИ КОНСТРУИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Существуют 3 стратегии конструирования ПО:

- *однократный проход* (водопадная стратегия) – линейная последовательность этапов конструирования;
- *инкрементная стратегия*. В начале процесса определяются все пользовательские и системные требования, оставшаяся часть конструирования выполняется в виде последовательности версий. Первая версия реализует часть запланированных возможностей, следующая версия реализует дополнительные возможности и т.д., пока не будет получена полная система;
- *эволюционная стратегия*. Система также строится в виде последовательности версий, но в начале процесса определены не все требования. Требования уточняются в результате разработки версий.

Характеристики стратегий конструирования ПО в соответствии с требованиями стандарта IEEE/EIA 12207.2 приведены в табл. 4.1.

4.1. Характеристики стратегий конструирования

Стратегия конструирования	В начале процесса определены все требования?	Множество циклов конструирования?	Промежуточное ПО распространяется?
Однократный проход	Да	Нет	Нет
Инкрементная (запланированное улучшение продукта)	Да	Да	Может быть
Эволюционная	Нет	Да	Да

ИНКРЕМЕНТНАЯ МОДЕЛЬ

Инкрементная модель является классическим примером инкрементной стратегии конструирования (рис. 4.4). Она объединяет элементы последовательной водопадной модели с итерационной философией макетирования.

Каждая линейная последовательность здесь вырабатывает поставляемый инкремент ПО. Например, ПО для обработки слов в 1-м инкременте реализует функции базовой обработки файлов, функции редактирования и документирования; во 2-м инкременте – более сложные возможности редактирования и документирования; в 3-м инкременте – проверку орфографии и грамматики; в 4-м инкременте – возможности компоновки страницы.

Первый инкремент приводит к получению базового продукта, реализующего базовые требования (правда, многие вспомогательные требования остаются нереализованными).

План следующего инкремента предусматривает модификацию базового продукта, обеспечивающую дополнительные характеристики и функциональность.

По своей природе инкрементный процесс итеративен, но в отличие от макетирования, инкрементная модель обеспечивает на каждом инкременте работающий продукт.

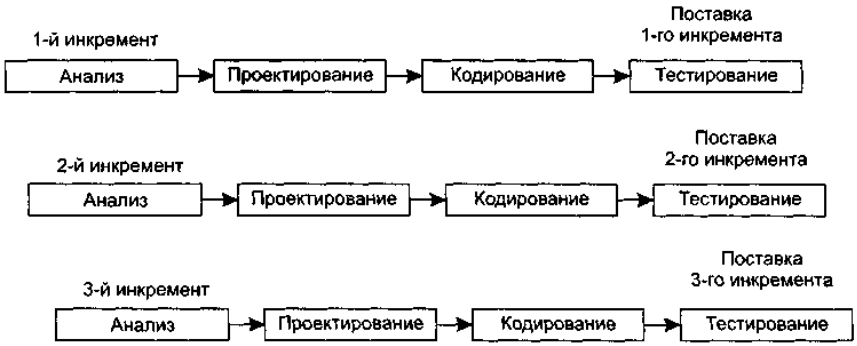

Рис. 4.4. Инкрементная модель

БЫСТРАЯ РАЗРАБОТКА ПРИЛОЖЕНИЙ

Модель быстрой разработки приложений RAD (Rapid Application Development) – второй пример применения инкрементной стратегии конструирования (рис. 4.5).

RAD-модель обеспечивает экстремально короткий цикл разработки. RAD – высокоскоростная адаптация линейной последовательной модели,

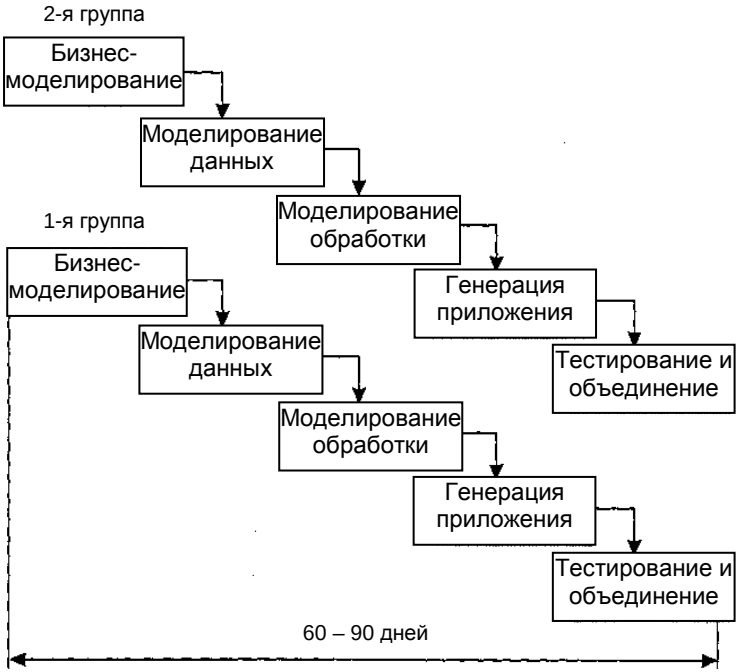

Рис. 4.5. Модель быстрой разработки приложений

в которой быстрая разработка достигается за счёт использования компонентно-ориентированного конструирования. Если требования полностью определены, а проектная область ограничена, RAD-процесс позволяет группе создать полностью функциональную систему за очень короткое время (60 – 90 дней). RAD-подход ориентирован на разработку информационных систем и выделяет следующие этапы:

- **бизнес-моделирование.** Моделируется информационный поток между бизнес-функциями. Ищется ответ на следующие вопросы: какая информация руководит бизнес-процессом? Какая информация генерируется? Кто генерирует её? Где информация применяется? Кто обрабатывает её?
- **моделирование данных**. Информационный поток, определённый на этапе бизнес-моделирования, отображается в набор объектов данных, которые требуются для поддержки бизнеса. Идентифицируются характеристики (свойства, атрибуты) каждого объекта, определяются отношения между объектами;
- **моделирование обработки.** Определяются преобразования объектов данных, обеспечивающие реализацию бизнес-функций. Создаются описания обработки для добавления, модификации, удаления или нахождения (исправления) объектов данных;

- **генерация приложения.** Предполагается использование методов, ориентированных на языки программирования 4-го поколения. Вместо создания ПО с помощью языков программирования 3-го поколения, RAD-процесс работает с повторно используемыми программными компонентами или создаёт повторно используемые компоненты. Для обеспечения конструирования используются утилиты автоматизации;
- **тестирование и объединение.** Поскольку применяются повторно используемые компоненты, многие программные элементы уже протестированы. Это уменьшает время тестирования (хотя все новые элементы должны быть протестированы).

Применение RAD возможно в том случае, когда каждая главная функция может быть завершена за 3 месяца. Каждая главная функция адресуется отдельной группе разработчиков, а затем интегрируется в целую систему.

Применение RAD имеет и свои недостатки, и ограничения.

1. Для больших проектов в RAD требуются существенные людские ресурсы (необходимо создать достаточное количество групп).
2. RAD применима только для таких приложений, которые могут декомпозироваться на отдельные модули и в которых производительность не является критической величиной.
3. RAD не применима в условиях высоких технических рисков (т.е. при использовании новой технологии).

СПИРАЛЬНАЯ МОДЕЛЬ

Спиральная модель – классический пример применения эволюционной стратегии конструирования.

Спиральная модель (автор Барри Боэм, 1988) базируется на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент – анализ риска, отсутствующий в этих парадигмах [22].

Как показано на рисунке 4.6, модель определяет четыре действия, представляемые четырьмя квадрантами спирали.

1. Планирование – определение целей, вариантов и ограничений.
2. Анализ риска – анализ вариантов и распознавание/выбор риска.
3. Конструирование – разработка продукта следующего уровня.
4. Оценивание – оценка заказчиком текущих результатов конструирования.

Интегрирующий аспект спиральной модели очевиден при учёте радиального измерения спирали. С каждой итерацией по спирали (продвижением от центра к периферии) строятся всё более полные версии ПО.

В первом витке спирали определяются начальные цели, варианты и ограничения, распознаётся и анализируется риск. Если анализ риска показывает неопределённость требований, на помощь разработчику и заказчику приходит макетирование (используемое в квадранте конструирования).

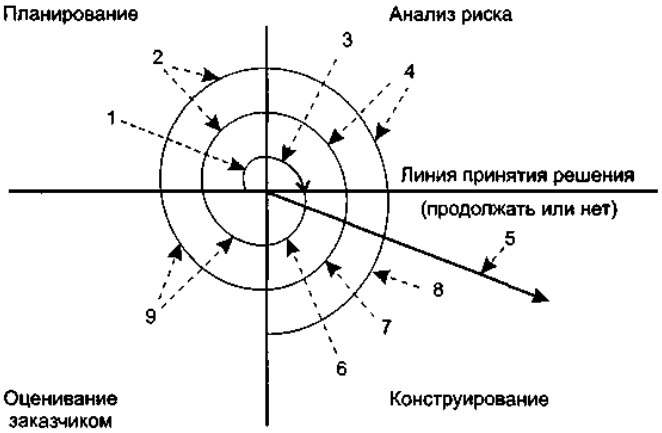

Рис. 4.6. Спиральная модель:
1 – начальный сбор требований и планирование проекта; *2* – та же работа, но на основе рекомендаций заказчика; *3* – анализ риска на основе начальных требований; *4* – анализ риска на основе реакции заказчика; *5* – переход к комплексной системе; *6* – начальный макет системы; *7* – следующий уровень макета; *8* – сконструированная система; *9* – оценивание заказчиком

Для дальнейшего определения проблемных и уточнённых требований может быть использовано моделирование. Заказчик оценивает инженерную (конструкторскую) работу и вносит предложения по модификации (квадрант оценки заказчиком). Следующая фаза планирования и анализа риска базируется на предложениях заказчика. В каждом цикле по спирали результаты анализа риска формируются в виде «продолжать, не продолжать». Если риск слишком велик, проект может быть остановлен.

В большинстве случаев движение по спирали продолжается, с каждым шагом продвигая разработчиков к более общей модели системы. В каждом цикле по спирали требуется конструирование (нижний правый квадрант), которое может быть реализовано классическим жизненным циклом или макетированием. Заметим, что количество действий по разработке (происходящих в правом нижнем квадранте) возрастает по мере продвижения от центра спирали.

Достоинства спиральной модели:

1) наиболее реально (в виде эволюции) отображает разработку программного обеспечения;

2) позволяет явно учитывать риск на каждом витке эволюции разработки;

3) включает шаг системного подхода в итерационную структуру разработки;

4) использует моделирование для уменьшения риска и совершенствования программного изделия.

Недостатки спиральной модели:

1) новизна (отсутствует достаточная статистика эффективности модели);

2) повышенные требования к заказчику;

3) трудности контроля и управления временем разработки.

КОМПОНЕНТНО-ОРИЕНТИРОВАННАЯ МОДЕЛЬ

Компонентно-ориентированная модель является развитием спиральной модели и тоже основывается на эволюционной стратегии конструирования. В этой модели конкретизируется содержание квадранта конструирования – оно отражает тот факт, что в современных условиях новая разработка должна основываться на повторном использовании существующих программных компонентов (рис. 4.7).

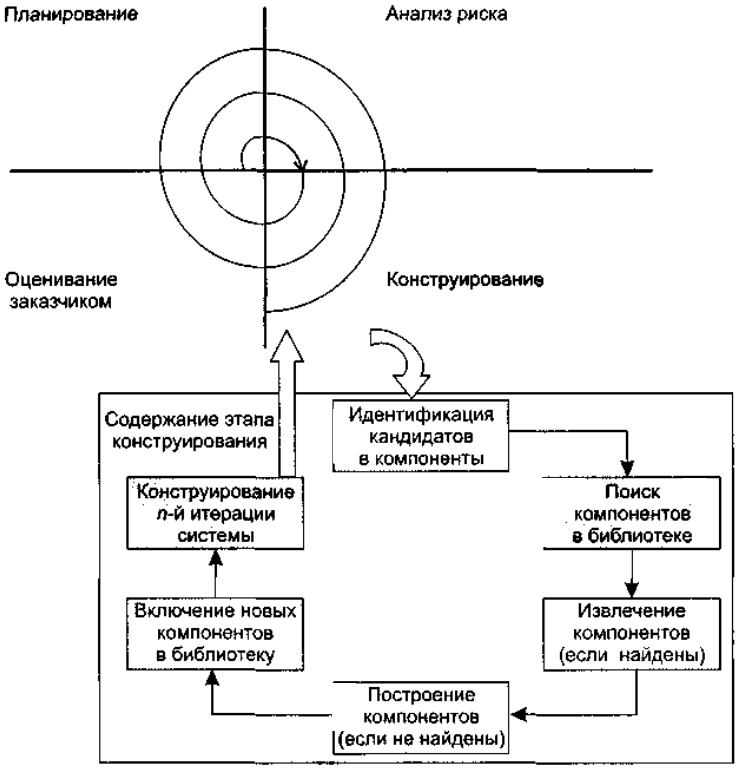

Рис. 4.7. Компонентно-ориентированная модель

Программные компоненты, созданные в реализованных программных проектах, хранятся в библиотеке. В новом программном проекте, исходя из требований заказчика, выявляются кандидаты в компоненты. Далее проверяется наличие этих кандидатов в библиотеке. Если они найдены, то компоненты извлекаются из библиотеки и используются повторно. В противном случае создаются новые компоненты, они применяются в проекте и включаются в библиотеку.

Достоинства компонентно-ориентированной модели:

1) уменьшает на 30% время разработки программного продукта;
2) уменьшает стоимость программной разработки до 70%;
3) увеличивает в полтора раза производительность разработки.

ТЯЖЕЛОВЕСНЫЕ И ОБЛЕГЧЁННЫЕ ПРОЦЕССЫ

Традиционно для упорядочения и ускорения программных разработок предлагались строго упорядочивающие тяжеловесные (heavyweight) процессы. В этих процессах прогнозируется весь объём предстоящих работ, поэтому они называются прогнозирующими (predictive) процессами. Порядок, который должен выполнять при этом человек-разработчик, чрезвычайно строг.

В последние годы появилась группа новых, облегчённых (lightweight) процессов [23]. Теперь их называют подвижными (agile) процессами. Они привлекательны отсутствием бюрократизма, характерного для тяжеловесных (прогнозирующих) процессов. Новые процессы должны воплотить в жизнь разумный компромисс между слишком строгой дисциплиной и полным её отсутствием. Иначе говоря, порядка в процессах разработки достаточно для того, чтобы получить разумную отдачу от разработчиков.

Подвижные процессы требуют меньшего объёма документации и ориентированы на человека. В них явно указано на необходимость использования природных качеств человеческой натуры (а не на применение действий, направленных наперекор этим качествам).

Более того, подвижные процессы учитывают особенности современного заказчика, а именно частые изменения его требований к программному продукту. Известно, что для прогнозирующих процессов частые изменения требований подобны смерти. В отличие от них, подвижные процессы адаптируют изменения требований и даже выигрывают от этого. Словом, подвижные процессы имеют адаптивную природу.

Таким образом, в современной инфраструктуре программной инженерии существуют два семейства процессов разработки:

- семейство прогнозирующих (тяжеловесных) процессов;
- семейство адаптивных (подвижных, облегчённых) процессов.

У каждого семейства есть свои достоинства, недостатки и область применения:

- адаптивный процесс используют при частых изменениях требований, малочисленной группе высококвалифицированных разработчиков и грамотном заказчике, который согласен участвовать в разработке;
- прогнозирующий процесс применяют при фиксированных требованиях и многочисленной группе разработчиков разной квалификации.

XP-ПРОЦЕСС

Экстремальное программирование (eXtreme Programming, XP) – облегчённый (подвижный) процесс (или методология) [24]. XP-процесс ориентирован на группы малого и среднего размера, строящие программное обеспечение в условиях неопределенных или быстро изменяющихся требований. XP-группу образуют до 10 сотрудников, которые размещаются в одном помещении.

Основная идея XP – устранить высокую стоимость изменения, характерную для приложений с использованием объектов, паттернов* и реляционных баз данных. Поэтому XP-процесс должен быть высокодинамичным процессом. XP-группа имеет дело с изменениями требований на всём протяжении итерационного цикла разработки, причём цикл состоит из очень коротких итераций. Четырьмя базовыми действиями в XP-цикле являются: кодирование, тестирование, выслушивание заказчика и проектирование. Динамизм обеспечивается с помощью четырёх характеристик: непрерывной связи с заказчиком (и в пределах группы), простоты (всегда выбирается минимальное решение), быстрой обратной связи (с помощью модульного и функционального тестирования), смелости в проведении профилактики возможных проблем.

Большинство принципов, поддерживаемых в XP (минимальность, простота, эволюционный цикл разработки, малая длительность итерации, участие пользователя, оптимальные стандарты кодирования и т.д.), продиктованы здравым смыслом и применяются в любом упорядоченном процессе. Просто в XP эти принципы, как показано в табл. 4.2, достигают «экстремальных значений».

4.2. Экстремумы в экстремальном программировании

Практика здравого смысла	XP-экстремум	XP-реализация
Проверки кода	Код проверяется всё время	Парное программирование
Тестирование	Тестирование выполняется всё время, даже с помощью заказчиков	Тестирование модуля, функциональное тестирование

Продолжение табл. 4.2

Практика здравого смысла	XP-экстремум	XP-реализация
Проектирование	Проектирование является частью ежедневной деятельности каждого разработчика	Реорганизация (refactoring)
Простота	Для системы выбирается простейшее проектное решение, поддерживающее её текущую функциональность	Самая простая вещь, которая могла бы работать
Архитектура	Каждый постоянно работает над уточнением архитектуры	Метафора
Тестирование интеграции	Интегрируется и тестируется несколько раз в день	Непрерывная интеграция
Короткие итерации	Итерации являются предельно короткими, продолжаются секунды, минуты, часы, а не недели, месяцы или годы	Игра планирования

Тот, кто принимает принцип «минимального решения» за хакерство, ошибается, в действительности XP – строго упорядоченный процесс. Простые решения, имеющие высший приоритет, в настоящее время рассматриваются как наиболее ценные части системы, в отличие от проектных решений, которые пока не нужны, а могут (в условиях изменения требований и операционной среды) и вообще не понадобиться.

Базис XP образуют перечисленные ниже двенадцать методов.

1. Игра планирования (Planning game) – быстрое определение области действия следующей реализации путём объединения деловых приоритетов и технических оценок. Заказчик формирует область действия, приоритетность и сроки с точки зрения бизнеса, а разработчики оценивают и прослеживают продвижение (прогресс).

2. Частая смена версий (Small releases) – быстрый запуск в производство простой системы. Новые версии реализуются в очень коротком (двухнедельном) цикле.

3. Метафора (Metaphor) – вся разработка проводится на основе простой, общедоступной истории о том, как работает вся система.

4. Простое проектирование (Simple design) – проектирование выполняется настолько просто, насколько это возможно в данный момент.

5. Тестирование (Testing) – непрерывное написание тестов для модулей, которые должны выполняться безупречно; заказчики пишут тесты для демонстрации законченности функций. «Тестируй, а затем кодируй» означает, что входным критерием для написания кода является «отказавший» тестовый вариант.

6. Реорганизация (Refactoring) – система реструктурируется, но её поведение не изменяется; цель – устранить дублирование, улучшить взаимодействие, упростить систему или добавить в неё гибкость.

7. Парное программирование (Pair programming) – весь код пишется двумя программистами, работающими на одном компьютере.

8. Коллективное владение кодом (Collective ownership) – любой разработчик может улучшать любой код системы в любое время.

9. Непрерывная интеграция (Continuous integration) – система интегрируется и строится много раз в день по мере завершения каждой задачи. Непрерывное регрессионное тестирование, т.е. повторение предыдущих тестов гарантирует, что изменения требований не приведут к регрессу функциональности.

10. 40-часовая неделя (40-hour week) – как правило, работают не более 40 часов в неделю. Нельзя удваивать рабочую неделю за счёт сверхурочных работ.

11. Локальный заказчик (On-site customer) – в группе всё время должен находиться представитель заказчика, действительно готовый отвечать на вопросы разработчиков.

12. Стандарты кодирования (Coding standards) – должны выдерживаться правила, обеспечивающие одинаковое представление программного кода во всех частях программной системы.

Игра планирования и частая смена версий зависят от заказчика, обеспечивающего набор «историй» (коротких описаний), характеризующих работу, которая будет выполняться для каждой версии системы. Версии генерируются каждые две недели, поэтому разработчики и заказчик должны прийти к соглашению о том, какие истории будут осуществлены в пределах двух недель. Полную функциональность, требуемую заказчику, характеризует пул историй; но для следующей двухнедельной итерации из пула выбирается подмножество историй, наиболее важное для заказчика. В любое время в пул могут быть добавлены новые истории, таким образом, требования могут быстро изменяться. Однако процессы двухнедельной генерации основаны на наиболее важных функциях, входящих в текущий пул, следовательно, изменчивость управляется. Локальный заказчик обеспечивает поддержку этого стиля итерационной разработки.

«Метафора» обеспечивает глобальное «видение» проекта. Она могла бы рассматриваться как высокоуровневая архитектура, но XP подчёркивает желательность проектирования при минимизации проектной докумен-

тации. Точнее говоря, XP предлагает непрерывное перепроектирование (с помощью реорганизации), при котором нет нужды в детализированной проектной документации, а для инженеров сопровождения единственным надёжным источником информации является программный код. Обычно после написания кода проектная документация выбрасывается. Проектная документация сохраняется только в том случае, когда заказчик временно теряет способность придумывать новые истории. Тогда систему помещают в «нафталин» и пишут руководство страниц на пять-десять по «нафталиновому» варианту системы. Использование реорганизации приводит к реализации простейшего решения, удовлетворяющего текущую потребность. Изменения в требованиях заставляют отказываться от всех «общих решений».

Парное программирование – один из наиболее спорных методов в XP, оно влияет на ресурсы, что важно для менеджеров, решающих, будет ли проект использовать XP. Может показаться, что парное программирование удваивает ресурсы, но исследования доказали: парное программирование приводит к повышению качества и уменьшению времени цикла. Для согласованной группы затраты увеличиваются на 15%, а время цикла сокращается на 40…50%. Для Интернет-среды увеличение скорости продаж покрывает повышение затрат. Сотрудничество улучшает процесс решения проблем, улучшение качества существенно снижает затраты сопровождения, которые превышают стоимость дополнительных ресурсов по всему циклу разработки.

Коллективное владение означает, что любой разработчик может изменять любой фрагмент кода системы в любое время. Непрерывная интеграция, непрерывное регрессионное тестирование и парное программирование XP обеспечивают защиту от возникающих при этом проблем.

«Тестируй, а затем кодируй» – эта фраза выражает акцент XP на тестировании. Она отражает принцип, по которому сначала планируется тестирование, а тестовые варианты разрабатываются параллельно анализу требований, хотя традиционный подход состоит в тестировании «чёрного ящика». Размышление о тестировании в начале цикла жизни – хорошо известная практика конструирования ПО (правда, редко осуществляемая практически).

Основным средством управления XP является метрика, а среда метрик – «большая визуальная диаграмма». Обычно используют 3–4 метрики, причём такие, которые видимы всей группе. Рекомендуемой в XP метрикой является «скорость проекта» – количество историй заданного размера, которые могут быть реализованы в итерации.

При принятии XP рекомендуется осваивать его методы по одному, каждый раз выбирая метод, ориентированный на самую трудную проблему группы. Конечно, все эти методы являются «не более чем правилами» – группа может в любой момент поменять их (если её сотрудники достигли

принципиального соглашения по поводу внесённых изменений). Защитники XP признают, что XP оказывает сильное социальное воздействие, и не каждый может принять его. Вместе с тем, XP – это методология, обеспечивающая преимущества только при использовании законченного набора базовых методов.

Рассмотрим структуру «идеального» XP-процесса. Основным структурным элементом процесса является XP-реализация, в которую многократно вкладывается базовый элемент – XP-итерация. В состав XP-реализации и XP-итерации входят три фазы – исследование, блокировка, регулирование. Исследование (exploration) – это поиск новых требований (историй, задач), которые должна выполнять система. Блокировка (commitment) – выбор для реализации конкретного подмножества из всех возможных требований (иными словами, планирование). Регулирование (steering) – проведение разработки, воплощение плана в жизнь.

XP рекомендует: первая реализация должна иметь длительность 2 – 6 месяцев, продолжительность остальных реализаций – около двух

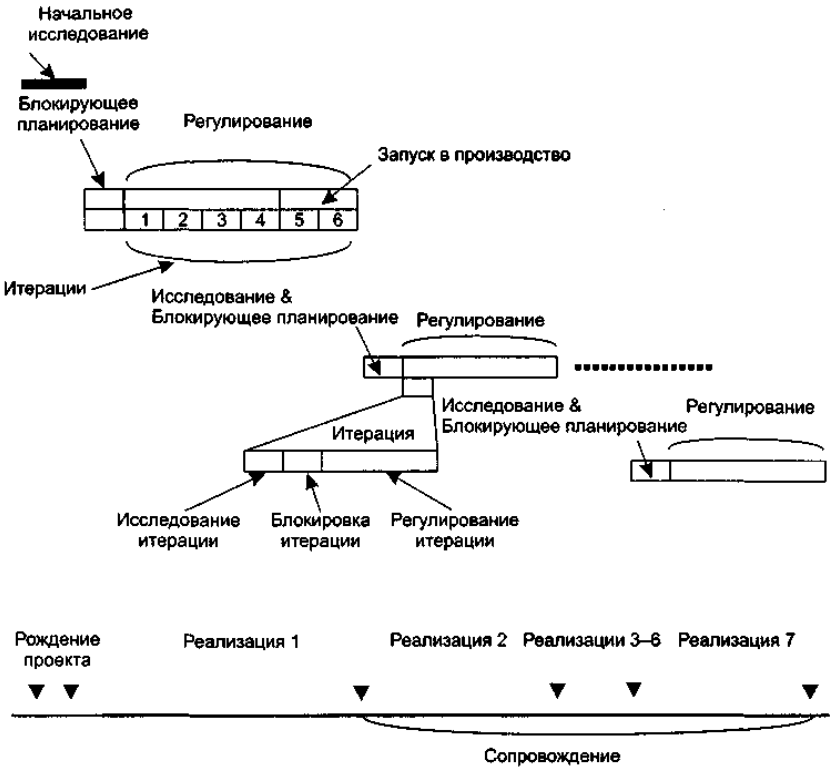

Рис. 4.8. Идеальный XP-процесс

месяцев, каждая итерация длится приблизительно две недели, а численность группы разработчиков не превышает 10 человек. XP-процесс для проекта с семью реализациями, осуществляемый за 15 месяцев, показан на рис. 4.8.

Процесс инициируется начальной исследовательской фазой.

Фаза исследования, с которой начинается любая реализация и итерация, имеет клапан «пропуска», на этой фазе принимается решение о целесообразности дальнейшего продолжения работы.

Предполагается, что длительность первой реализации составляет 3 месяца, длительность второй – седьмой реализаций – 2 месяца. Вторая – седьмая реализации образуют период сопровождения, характеризующий природу XP-проекта. Каждая итерация длится две недели, за исключением тех, которые относят к поздней стадии реализации – «запуску в производство» (в это время темп итерации ускоряется).

Наиболее трудна первая реализация – пройти за три месяца от обычного старта (скажем, отдельный сотрудник не зафиксировал никаких требований, не определены ограничения) к поставке заказчику системы промышленного качества.

МОДЕЛИ КАЧЕСТВА ПРОЦЕССОВ КОНСТРУИРОВАНИЯ

В современных условиях жёсткой конкуренции очень важно гарантировать высокое качество процесса конструирования ПО. Такую гарантию даёт сертификат качества процесса, подтверждающий его соответствие принятым международным стандартам. Каждый такой стандарт фиксирует свою модель обеспечения качества. Наиболее авторитетны модели стандартов ISO 9001:2000, ISO/ IEC 15504 и модель зрелости процесса конструирования ПО (Capability Maturity Model – CMM) Института программной инженерии при американском университете Карнеги-Меллон.

Модель стандарта ISO 9001:2000 ориентирована на процессы разработки из любых областей человеческой деятельности. Стандарт ISO/ IEC 15504 специализируется на процессах программной разработки и отличается более высоким уровнем детализации. Достаточно сказать, что объём этого стандарта превышает 500 страниц. Значительная часть идей ISO/IEC 15504 взята из модели CMM.

Базовым понятием модели CMM считается *зрелость* компании. Незрелой называют компанию, где процесс конструирования ПО и принимаемые решения зависят только от таланта конкретных разработчиков. Как следствие, здесь высока вероятность превышения бюджета или срыва сроков окончания проекта.

Напротив, в зрелой компании работают ясные процедуры управления проектами и построения программных продуктов. По мере необходимо-

сти эти процедуры уточняются и развиваются. Оценки длительности и затрат разработки точны, основываются на накопленном опыте. Кроме того, в компании имеются и действуют корпоративные стандарты на процессы взаимодействия с заказчиком, процессы анализа, проектирования, программирования, тестирования и внедрения программных продуктов. Всё это создаёт среду, обеспечивающую качественную разработку программного обеспечения.

Таким образом, модель CMM фиксирует критерии для оценки зрелости компании и предлагает рецепты для улучшения существующих в ней процессов. Иными словами, в ней не только сформулированы условия, необходимые для достижения минимальной организованности процесса, но и даются рекомендации по дальнейшему совершенствованию процессов.

Очень важно отметить, что модель CMM ориентирована на построение системы постоянного улучшения процессов. В ней зафиксированы 5 уровней зрелости (рис. 4.9) и предусмотрен плавный, поэтапный подход к совершенствованию процессов – можно поэтапно получать подтверждения об улучшении процессов после каждого уровня зрелости.

Начальный уровень (уровень 1) означает, что процесс в компании не формализован. Он не может строго планироваться и отслеживаться, его успех носит случайный характер. Результат работы целиком и полностью

Рис. 4.9. Пять уровней зрелости модели CMM

зависит от личных качеств отдельных сотрудников. При увольнении таких сотрудников проект останавливается.

Для перехода на **повторяемый** уровень (уровень 2) необходимо внедрить формальные процедуры для выполнения основных элементов процесса конструирования. Результаты выполнения процесса соответствуют заданным требованиям и стандартам. Основное отличие от уровня 1 состоит в том, что выполнение процесса планируется и контролируется. Применяемые средства планирования и управления дают возможность повторения ранее достигнутых успехов.

Следующий, **определённый** уровень (уровень 3) требует, чтобы все элементы процесса были определены, стандартизованы и задокументированы. Основное отличие от уровня 2 заключается в том, что элементы процесса уровня 3 планируются и управляются на основе единого стандарта компании. Качество разрабатываемого ПО уже не зависит от способностей отдельных личностей.

С переходом на **управляемый** уровень (уровень 4) в компании принимаются количественные показатели качества как программных продуктов, так и процесса. Это обеспечивает более точное планирование проекта и контроль качества его результатов. Основное отличие от уровня 3 состоит в более объективной, количественной оценке продукта и процесса.

Высший, **оптимизирующий** уровень (уровень 5) подразумевает, что главной задачей компании становится постоянное улучшение и повышение эффективности существующих процессов, ввод новых технологий. Основное отличие от уровня 4 заключается в том, что технология создания и сопровождения программных продуктов планомерно и последовательно совершенствуется.

Каждый уровень CMM характеризуется *областью ключевых процессов* (ОКП), причём считается, что каждый последующий уровень включает в себя все характеристики предыдущих уровней. Иначе говоря, для 3-го уровня зрелости рассматриваются ОКП 3-го уровня, ОКП 2-го уровня и ОКП 1-го уровня. Область ключевых процессов образуют процессы, которые при совместном выполнении приводят к достижению определённого набора целей. Например, ОКП 5-го уровня образуют процессы:

- предотвращения дефектов;
- управления изменениями технологии;
- управления изменениями процесса.

Если все цели ОКП достигнуты, компании присваивается сертификат данного уровня зрелости. Если хотя бы одна цель не достигнута, то компания не может соответствовать данному уровню CMM.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Дайте определение технологии конструирования программного обеспечения.
2. Какие этапы классического жизненного цикла вы знаете?
3. Охарактеризуйте содержание этапов классического жизненного цикла.
4. Объясните достоинства и недостатки классического жизненного цикла.
5. Чем отличается классический жизненный цикл от макетирования?
6. Какие существуют формы макетирования?
7. Чем отличаются друг от друга стратегии конструирования ПО?
8. Укажите сходства и различия классического жизненного цикла и инкрементной модели.
9. Объясните достоинства и недостатки инкрементной модели.
10. Чем отличается модель быстрой разработки приложений от инкрементной модели?
11. Объясните достоинства и недостатки модели быстрой разработки приложений.
12. Укажите сходства и различия спиральной модели и классического жизненного цикла.
13. В чём состоит главная особенность спиральной модели?
14. Чем отличается компонентно-ориентированная модель от спиральной модели и классического жизненного цикла?
15. Перечислите достоинства и недостатки компонентно-ориентированной модели.
16. Чем отличаются тяжеловесные процессы от облегчённых процессов?
17. Чем отличаются тяжеловесные процессы от прогнозирующих процессов?
18. Чем отличаются подвижные процессы от облегчённых процессов?
19. Перечислите достоинства и недостатки тяжеловесных процессов.
20. Перечислите достоинства и недостатки облегчённых процессов.
21. Приведите примеры тяжеловесных процессов.
22. Приведите примеры облегчённых процессов.
23. Перечислите характеристики XP-процесса.
24. Перечислите методы XP-процесса.
25. В чём состоит главная особенность XP-процесса?
26. Охарактеризуйте содержание игры планирования в XP-процессе.

27. Охарактеризуйте назначение метафоры в XP-процессе.
28. Какова особенность проектирования в XP-процессе?
29. Какова особенность программирования в XP-процессе?
30. Что такое реорганизация?
31. Что такое коллективное владение?
32. Какова особенность тестирования в XP-процессе?
33. Чем отличается XP-реализация от XP-итерации?
34. Чем XP-реализация похожа на XP-итерацию?
35. Какова длительность XP-реализации?
36. Какова длительность XP-итерации?
37. Какова максимальная численность группы XP-разработчиков?
38. Какие модели качества процессов конструирования вы знаете?
39. Охарактеризуйте модель CMM.
40. Охарактеризуйте уровень зрелости знакомой вам фирмы.

ЗАКЛЮЧЕНИЕ

В учебном пособии изложен материал, раскрывающий важные аспекты разработки программной составляющей практически любой вычислительной системы. Программные системы являются наиболее динамично развивающимися и используются конечным пользователем в повседневной жизни.

Материалы, изложенные в работе, позволяют не только понять сущность процесса разработки программного обеспечения, но и использовать в реальных проектах.

СПИСОК ЛИТЕРАТУРЫ

1. Myers, G. Composite Structured Design / G. Myers. – NY : Van Nostrand Reinhold, 1978.

2. Parnas, D. On the Criteria to the Be Used in Decomposing Systems into Modules / D. Parnas // Communications of the ACM. – 1972. – Vol. 15(12). – P. 1053 – 1058.

3. Page-Jones, M. The Practical Guide to Structured Systems Design / M. Page-Jones // Englewood Cliffs. – NY : Yourdon Press, 1988.

4. Vliet, J.C. van. Software Engineering: Principles and Practice / J.C. Vliet, van. – John Wiley & Sons, 1993. – 558 p.

5. Yourdon, E. Structured Design: fundamentals of a discipline of computer program and systems design / E. Yourdon, L. Constantine // Englewood Cliffs. – NJ : Prentice-Hall, 1979.

6. Halstead, M.H. Elements of Software Science / M.H. Halstead. – NY : Elsevier North-Holland, 1977.

7. McCabe, T.J. A Complexity Measure / T.J. McCabe // IEEE Transactions on Software Engineering. – 1976. – Vol. 2, N. 4. – P. 308 – 320.

8. Fenton, N.E. Software Metrics: A Rigorous & Practical Approach. 2nd edition / N.E. Fenton, S.L. Pfleeger // International Thomson Computer Press. – 1997. – 647 p.

9. Oviedo, E.I. Control Flow, Data Flow and Program Complexity / E.I. Oviedo // Proc. IEEE COMPSAC. – 1980. – November. – P. 146 – 152.

10. Henry, S. Software Structure Metrics Based on Information Flow / S. Henry, D. Kafura // IEEE Transactions on Software Engineering. – 1981. – Vol. 7, N. 5. – P. 510 – 518.

11. Booch, G. Object-Oriented analysis and design. 2nd edition / G. Booch. – Addison-Wesley, 1994. – 590 p.

12. Graham, I. Object-Oriented Methods. Principles & Practice. 3rd edition / I. Graham. – Addison-Wesley, 2001. – 853 p.

13. Jacobcon, I. Object-Oriented Software Engineering / I. Jacobcon, M. Christerson, P. Jonsson, G.J. Overgaard. – Addison-Wesley, 1993. – 528 p.

14. Page-Jones, M. Fundamentals of Object-Oriented Design in UML / M. Page-Jones. – Addison-Wesley, 2001. – 479 p.

15. Pressman, R.S. Software Engineering: A Practioner's Approach. 5th edition. / R.S. Pressman. – McGraw-Hill, 2000. – 943 p.

16. Rumbaugh, J. Object Oriented Modeling and Design. Prentice Hall / J. Rumbaugh, M. Blaha, W. Premerlani. – 1991. – 500 p.

17. OMG Unified Modeling Language Specif ication. Version 1.4. Object Management Group, Inc., 2001. – 566 p.

18. Booch, G. The Unified Modeling Language User Guide / G. Booch, J. Rumbaugh, I. Jacobcon. – Addison-Wesley, 1999. – 483 p.

19. Rumbaugh, J. The Unified Modeling Language Reference Manual / J. Rumbaugh, I. Jacobcon, G. Booch. – Addison-Wesley, 1999. – 567 p.

20. Орлов, С.А. Технологии разработки программного обеспечения : учебник / С.А. Орлов. – СПб. : Изд-во «Питер-принт», 2002. – 322 с.

21. Royce, Walker W. Managing the development of large software systems: concepts and techniques / Walker W. Royce // Proc. IEEE WESTCON. – Los Angeles. – 1970. – August. – P. 1 – 9.

22. Boehm, B.W. A spiral model of software development and enhancement / B.W. Boehm // IEEE Computer. – 1988. – N 5. – P. 61 – 72.

23. Fowler, M. The New Methodology / M. Fowler. – URL : http://www.martinfowler.com, 2001.

24. Beck, K. Extreme Programming Explained. Embrace Change / K. Beck. – Addison-Wesley, 1999. – 211 p.

25. Гайсарян, С.С. Объектно-ориентированное проектирование, Центр информационных технологий / С.С. Гайсарян. – URL : http://www.citmgu.ru, 2008.

26. Голицына, О.Л. Программное обеспечение : учебное пособие / О.Л. Голицына, И.В. Попов, Т.Л. Партыка. – 3-е изд. – СПб. : Питер, 2004. – 528 с.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 3
1. ОСНОВЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ 4
Особенности процесса синтеза программных систем 4
Особенности этапа проектирования 5
Структурирование системы 6
Моделирование управления 8
Декомпозиция подсистем на модули 10
Модульность 10
Информационная закрытость 12
Связность модуля 13
Функциональная связность 14
Информационная связность 15
Коммуникативная связность 15
Процедурная связность 16
Временная связность 17
Логическая связность 18
Связность по совпадению 19
Определение связности модуля 19
Сцепление модулей 20
Сложность программной системы 22
Характеристики иерархической структуры программной системы 22
Контрольные вопросы 25
2. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРЕДСТАВЛЕНИЯ ПРОГРАММНЫХ СИСТЕМ 27
Принципы объектно-ориентированного представления программных систем 27
Абстрагирование 27
Инкапсуляция 28
Модульность 30
Иерархическая организация 30
Объекты 31
Общая характеристика объектов 31
Виды отношений между объектами 34
Связи 34

Видимость объектов 37
Агрегация 38
Классы 39
Общая характеристика классов 39
Виды отношений между классами 40
Ассоциации классов 41
Наследование 42
Полиморфизм 44
Агрегация 45
Зависимость 47
Конкретизация 48
Контрольные вопросы 50
3. БАЗИС ЯЗЫКА ВИЗУАЛЬНОГО МОДЕЛИРОВАНИЯ 51
Унифицированный язык моделирования 51
Предметы в UML 52
Отношения в UML 55
Диаграммы в UML 56
Механизмы расширения в UML 57
Контрольные вопросы 59
4. ОРГАНИЗАЦИЯ ПРОЦЕССА КОНСТРУИРОВАНИЯ 61
Определение технологии конструирования программного обеспечения 61
Классический жизненный цикл 62
Макетирование 64
Стратегии конструирования ПО 66
Инкрементная модель 67
Быстрая разработка приложений 67
Спиральная модель 69
Компонентно-ориентированная модель 71
Тяжеловесные и облегчённые процессы 72
XP-процесс 73
Модели качества процессов конструирования 78
Контрольные вопросы 81
ЗАКЛЮЧЕНИЕ 83
СПИСОК ЛИТЕРАТУРЫ 84

Учебное издание

МИЛОВАНОВ Игорь Викторович,
ЛОСКУТОВ Вячеслав Иванович

ОСНОВЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Учебное пособие

Редактор И.В. Калистратова
Инженер по компьютерному макетированию М.Н. Рыжкова

Подписано в печать 31.05.2011.
Формат 60 × 84 / 16. 5,11 усл. печ. л. Тираж 100 экз. Заказ № 235

Издательско-полиграфический центр ГОУ ВПО ТГТУ
392000, г. Тамбов, ул. Советская, д. 106, к. 14