Владимирский государственный университет

Д. А. ГРАДУСОВ А. В. ШУТОВ

ТЕОРЕТИЧЕСКИЕ ВОПРОСЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Владимир 2020

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

Д. А. ГРАДУСОВ А. В. ШУТОВ

ТЕОРЕТИЧЕСКИЕ ВОПРОСЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Электронное издание

Владимир 2020

ISBN 978-5-9984-1097-0

© Градусов Д. А., Шутов А. В., 2020

УДК 004.41
ББК 65с51
Г75

Рецензенты:
Доктор технических наук, профессор
зав. кафедрой информационных систем и программной инженерии
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
И. Е. Жигалов

Кандидат технических наук, доцент
доцент кафедры информационных технологий
Владимирского филиала Российской академии народного
хозяйства и государственной службы при Президенте РФ
С. В. Поляков

Градусов, Д. А. Теоретические вопросы разработки программного обеспечения : учеб. пособие / Д. А. Градусов, А. В. Шутов ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2020. – 171 с. – ISBN 978-5-9984-1097-0. – Системные требования: Intel от 1,3 ГГц; Windows 7/8/10; Adobe Reader; дисковод DVD-ROM. Объем 2,75 Мб. – Загл. с титул. экрана.

В пособии рассмотрены теоретические вопросы, касающиеся процесса разработки программного обеспечения, – история подходов к разработке ПО, существующие международные стандарты в области разработки программного обеспечения, различные подходы к организации жизненного цикла разработки (включая как классические, так и гибкие методологии разработки), подходы к тестированию ПО, методы оценки стоимости разрабатываемого ПО.

Предназначено для студентов вузов направления подготовки 09.03.03 «Прикладная информатика», может быть использовано при проведении занятий по дисциплинам «Программная инженерия», «Тестирование информационных систем», «Проектирование информационных систем», «Технологии программирования корпоративных информационных систем».

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Ил. 29. Табл. 16. Библиогр.: 29 назв.

ISBN 978-5-9984-1097-0 © Градусов Д. А., Шутов А. В., 2020

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ ... 5
Глава 1. ИСТОРИЯ РАЗВИТИЯ ТЕХНОЛОГИЙ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ... 6
1.1 Стихийное программирование ... 6
1.2 Структурное программирование ... 9
1.3 Объектно-ориентированное программирование ... 12
1.4 Компонентное программирование ... 14
Вопросы к главе 1 ... 20
Глава 2. СТАНДАРТИЗАЦИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ... 22
2.1 Основные понятия в области стандартизации ... 22
2.2 Стандарты в области управления качеством программного обеспечения ... 26
2.3 Стандарты в области управления проектами разработки программного обеспечения ... 27
2.4 Стандарты в области проектирования программного обеспечения ... 29
Вопросы к главе 2 ... 33
Глава 3. МЕТОДОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ... 34
3.1 Понятие методологии разработки ПО ... 34
3.2 Модель Waterfall ... 34
3.3 Итерационная модель ... 42
3.4 Спиральная модель ... 44
3.5 Agile ... 47
3.6 Методология Scrum ... 49
3.7 Методология XP ... 51
3.8 Методология RUP ... 57
3.9 Методология RAD ... 60
3.10 Некоторые другие методологии разработки ПО ... 63
Вопросы к главе 3 ... 69

Глава 4. ТЕСТИРОВАНИЕ КАК ЧАСТЬ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 70
4.1 Роль тестирования ПО 70
4.2 Принципы тестирования 74
4.3 Основной процесс тестирования 75
4.4 Уровни тестирования ПО 79
4.5 Организация и независимость тестирования 90
4.6 Мониторинг тестирования и контроль тестирования 91
Вопросы к главе 4 94
Глава 5. ОЦЕНКА В РАЗВИТИИ ПРОГРАММНОГО ПРОЕКТА 95
5.1 Роль оценки в планировании проекта 95
5.2 Неопределенность в оценке 101
5.3 Факторы, влияющие на оценку 108
5.4 Факторы, влияющие на выбор метода оценки 116
Вопросы к главе 5 121
Глава 6. МЕТОДЫ ОЦЕНКИ СТОИМОСТИ ПО 122
6.1 Метод оценки по аналогии 122
6.2 Метод параметрических оценок 123
6.3 Метод оценки «снизу вверх» 124
6.4 Методы, основанные на экспертных оценках 126
6.5 Линейный метод 131
6.6 Методы, основанные на функциональных пунктах 133
6.7 Методы COCOMO и COCOMO II 147
6.8 Модель Путнэма (SLIM) 159
6.9 Программные средства для оценки стоимости ПО 162
Вопросы к главе 6 165
ЗАКЛЮЧЕНИЕ 166
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 167

ВВЕДЕНИЕ

Разработка программного обеспечения представляет собой сложный многоступенчатый процесс, требующий согласованных усилий большого числа людей. Сложность процесса разработки ПО особенно ярко характеризует тот факт, что достаточно большой процент проектов по разработке программного обеспечения завершается с превышением бюджета или установленных сроков, либо не завершается вообще.

Подобных ситуаций можно избежать в результате грамотного менеджмента, то есть правильного планирования и организации процесса разработки программного обеспечения, а также предварительной оценки бюджета проекта и ее своевременной проекции.

Данные вопросы встают перед разработчиками программного обеспечения на протяжении последних 50 – 70 лет. Опыт успешной разработки ПО был обобщен и зафиксирован в ряде методологий и международных стандартов, охватывающих как весь процесс разработки программного обеспечения, так и отдельные его аспекты.

В пособии рассмотрены теоретические вопросы, касающиеся процесса разработки программного обеспечения, включая историю подходов к разработке ПО, существующие международные стандарты в области разработки программного обеспечения, различные подходы к организации жизненного цикла разработки (включая как классические, так и гибкие методологии разработки), подходы к тестированию ПО, методы оценки стоимости разрабатываемого ПО.

Глава 1. ИСТОРИЯ РАЗВИТИЯ ТЕХНОЛОГИЙ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1 Стихийное программирование

Этот этап охватывает период от момента появления первых вычислительных машин до середины 60-х годов XX в. В этот период практически отсутствовали сформулированные технологии и программирование фактически было искусством. Первые программы имели простейшую структуру. Они состояли из собственно программы на машинном языке и обрабатываемых ею данных (рис. 1.1).

Рис. 1.1 - Структура первых программ

Сложность программ в машинных кодах ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и местонахождение данных при программировании мнемоники кодов операций. В результате программы стали более «читаемыми».

Создание языков программирования высокого уровня, таких как FORTRAN и ALGOL, существенно упростило программирование вычислений, снизив уровень детализации операций. Это, в свою очередь, позволило увеличить сложность программ.

Революционным было появление в языках средств, позволяющих оперировать подпрограммами (идея написания подпрограмм появилась гораздо раньше, но отсутствие средств

поддержки в первых языковых средствах существенно снижало эффективность их применения.) Подпрограммы можно было сохранять и использовать в других программах. В результате были созданы огромные библиотеки расчетных и служебных подпрограмм, которые по мере надобности вызывались из разрабатываемой программы.

Типичная программа того времени состояла из основной программы, области ***глобальных данных*** и набора подпрограмм (в основном библиотечных), выполняющих обработку всех данных или их части (рис. 1.2).

Рис. 1.2 - Архитектура программы с глобальной областью данных

Слабым местом такой архитектуры было то, что при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой. Например, подпрограмма поиска корней уравнения на заданном интервале по методу деления отрезка пополам меняет величину интервала. Если при выходе из подпрограммы не предусмотреть восстановления первоначального интервала, то в глобальной области окажется неверное значение интервала. Чтобы сократить количество таких ошибок в подпрограммах, было предложено размещать ***локальные данные.***

Сложность разрабатываемого программного обеспечения при использовании подпрограмм с локальными данными по-прежнему

ограничивалась возможностью программиста отслеживать процессы обработки данных, но уже на новом уровне. Однако появление средств поддержки подпрограмм позволило осуществлять разработку программного обеспечения нескольким программистам параллельно.

В начале 60-х годов XX в. разразился «кризис программирования». Он выражался в том, что фирмы, взявшиеся за разработку сложного программного обеспечения, такого как операционные системы, срывали все сроки завершения проектов. Проект устаревал раньше, чем был готов к внедрению, увеличивалась его стоимость, и в результате многие проекты так никогда и не были завершены.

Объективно все это было вызвано несовершенством технологии программирования. Прежде всего стихийно использовалась разработка «снизу—вверх» — подход, при котором вначале проектировали и реализовывали сравнительно простые подпрограммы, из которых затем пытались построить сложную программу. В отсутствие четких моделей описания подпрограмм и методов их проектирования создание каждой подпрограммы превращалось в непростую задачу, интерфейсы подпрограмм получались сложными, и при сборке программного продукта выявлялось большое количество ошибок согласования. Исправление таких ошибок, как правило, требовало серьезного изменения уже разработанных подпрограмм, что еще более осложняло ситуацию, так как при этом в программу часто вносились новые ошибки, которые также необходимо было исправлять... В конечном итоге процесс тестирования и отладки программ занимал более 80 % времени разработки, если вообще когда-нибудь заканчивался. На повестке дня самым серьезным образом стоял вопрос разработки технологии создания сложных программных продуктов, снижающей вероятность ошибок проектирования.

Анализ причин возникновения большинства ошибок позволил сформулировать новый подход к программированию, который был назван «структурным».

1.2 Структурное программирование

Структурный подход к программированию появился в **60 — 70-е** годы XX в. ***Данный подход*** представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит ***декомпозиция*** (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших (до 40—50 операторов) подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т. д.) данный способ получил название ***процедурной*** декомпозиции.

В отличие от используемого ранее процедурного подхода к декомпозиции, структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры. Проектирование, таким образом, осуществлялось «сверху—вниз» и подразумевало реализацию общей идеи, обеспечивая проработку интерфейсов подпрограмм. Одновременно вводились ограничения на конструкции алгоритмов, рекомендовались формальные модели их описания, а также специальный метод проектирования алгоритмов — метод пошаговой детализации.

Поддержка принципов структурного программирования была заложена в основу так называемых ***процедурных*** языков программирования. Как правило, они включали основные «структурные» операторы передачи управления, поддерживали вложение подпрограмм, локализацию и ограничение области

«видимости» данных. Среди наиболее известных языков этой группы стоит назвать PL/1, ALGOL-68, Pascal, С.

Одновременно со структурным программированием появилось огромное количество языков, базирующихся на других концепциях, но большинство из них не выдержало конкуренции. Какие-то языки были просто забыты, идеи других были в дальнейшем использованы в следующих версиях развиваемых языков.

Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития ***структурирования данных.*** Как следствие этого в языках появляется возможность определения пользовательских типов данных. Одновременно усилилось стремление разграничить доступ к глобальным данным программы, чтобы уменьшить количество ошибок, возникающих при работе с глобальными данными. В результате появилась и начала развиваться технология модульного программирования.

Модульное программирование предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные в отдельно компилируемые ***модули*** (библиотеки подпрограмм), например, модуль графических ресурсов, модуль подпрограмм вывода на принтер. Связи между модулями при использовании данной технологии осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым «внутренним» переменным) запрещен (рис. 1.3). Эту технологию поддерживают современные версии языков Pascal и С (С++), языки Ада и Modula.

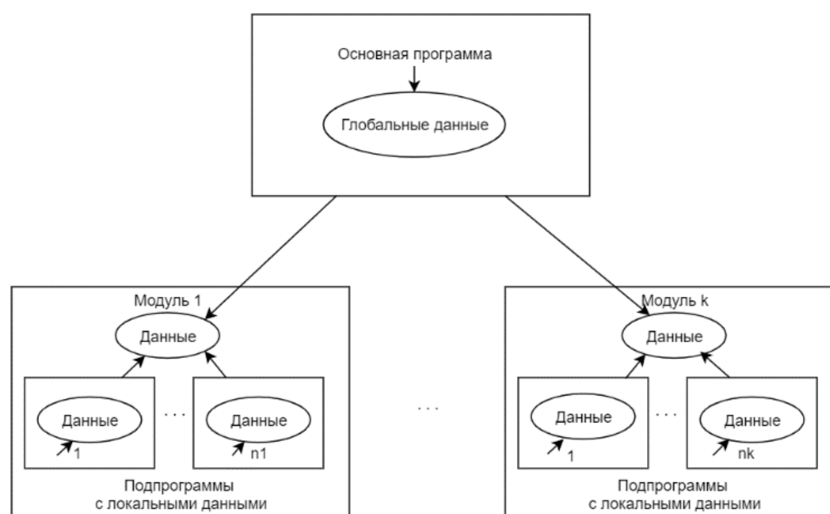

Рис. 1.3 - Архитектура программы, состоящей из модулей

Использование модульного программирования существенно упростило разработку программного обеспечения несколькими программистами. Теперь каждый из них мог разрабатывать свои модули независимо, обеспечивая взаимодействие модулей через специально оговоренные межмодульные интерфейсы. Кроме того, модули в дальнейшем без изменений можно было использовать в других разработках, что повысило производительность труда программистов.

Практика показала, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надежные программы, размер которых ***не превышает 100 ООО операторов***. Узким местом модульного программирования является то, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы (из-за раздельной компиляции модулей обнаружить эти ошибки раньше невозможно). При увеличении размера программы обычно возрастает сложность межмодульных интерфейсов, и с некоторого момента предусмотреть взаимовлияние отдельных частей программы становится практически невозможно. Для разработки программного обеспечения большого объема было предложено использовать ***объектный подход.***

1.3 Объектно-ориентированное программирование

Третий этап — объектный подход к программированию (с середины 80-х до конца 90-х годов XX в.). ***Объектно-ориентированное программирование*** определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности ***объектов,*** каждый из которых является экземпляром определенного типа ***(класса),*** а классы образуют иерархию с ***наследованием*** свойств. Взаимодействие программных объектов в такой системе осуществляется путем передачи ***сообщений***.

Объектная структура программы впервые была использована в языке имитационного моделирования сложных систем Simula, появившемся еще в 60-х годах XX в. Естественный для языков моделирования способ представления программы получил развитие в другом специализированном языке моделирования языке Smalltalk (70-е годы XX в.), а затем был использован в новых версиях универсальных языков программирования, таких как Pascal, C++, Modula, Java.

Основным достоинством объектно-ориентированного программирования по сравнению с модульным программированием является «более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку. Это приводит к более полной локализации данных и интегрированию их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) программ.

Кроме этого, объектный подход предлагает новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполнения. Эти механизмы позволяют конструировать сложные объекты из сравнительно простых. В результате существенно увеличивается показатель повторного

использования кодов и появляется возможность создания библиотек классов для различных применений.

Бурное развитие технологий программирования, основанных на объектном подходе, позволило решить многие проблемы. Так, были созданы среды, поддерживающие ***визуальное программирование,*** например, Delphi, C++ Builder, Visual C++ и т. д. При использовании визуальной среды у программиста появляется возможность проектировать некоторую часть, например, интерфейсы будущего продукта, с применением визуальных средств добавления и настройки специальных библиотечных компонентов. Результатом визуального проектирования является заготовка будущей программы, в которую уже внесены соответствующие коды.

Использование объектного подхода имеет много преимуществ, однако его конкретная реализация в объектно-ориентированных языках программирования, таких как Pascal и C++, имеет существенные недостатки:

1. фактически отсутствуют стандарты компоновки двоичных результатов компиляции объектов в единое целое даже в пределах одного языка программирования. Компоновка объектов, полученных разными компиляторами C++, в лучшем случае проблематична, что приводит к необходимости разработки программного обеспечения с использованием средств и возможностей одного языка программирования высокого уровня и одного компилятора, а значит, требует наличия исходных кодов используемых библиотек классов;
2. изменение реализации одного из программных объектов, как минимум, связано с перекомпиляцией соответствующего модуля и перекомпоновкой всего программного обеспечения, использующего данный объект.

Таким образом, при использовании этих языков программирования сохраняется зависимость модулей программного обеспечения от адресов экспортируемых полей и методов, а также

структур и форматов данных. Эта зависимость объективна, так как модули должны взаимодействовать между собой, обращаясь к ресурсам друг друга. Связи модулей нельзя разорвать, но можно попробовать стандартизировать их взаимодействие, на чем и основан компонентный подход к программированию.

1.4 Компонентное программирование

Четвертый этап — компонентный подход и CASE-**технологии (с середины 90-х годов** XX **в. до нашего времени).** ***Компонентный подход*** предполагает построение программного обеспечения из отдельных компонентов физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через ***стандартизованные двоичные интерфейсы.*** В отличие от обычных объектов объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию. На сегодня рынок объектов стал реальностью, так, в Интернете существуют узлы, предоставляющие большое количество компонентов, рекламой компонентов забиты журналы. Это позволяет программистам создавать продукты, хотя бы частично состоящие из повторно использованных частей, т. е. использовать технологию, хорошо зарекомендовавшую себя в области проектирования аппаратуры.

Компонентный подход лежит в основе технологий, разработанных на базе COM (Component Object Model — компонентная модель объектов), и технологии создания распределенных приложений CORBA (Common Object Request Broker Architecture — общая архитектура с посредником обработки запросов

объектов). Эти технологии используют сходные принципы и различаются лишь особенностями их реализации.

Технология COM фирмы Microsoft является развитием технологии OLE I (Object Linking and Embedding — связывание и внедрение объектов), которая использовалась в ранних версиях Windows для создания составных документов. Технология COM определяет ***общую парадигму взаимодействия программ любых типов:*** библиотек, приложений, операционной системы, т. е. позволяет одной части программного обеспечения использовать функции ***(службы),*** предоставляемые другой, независимо от того, функционируют ли эти части в пределах одного процесса, в разных процессах на одном компьютере или на разных компьютерах. Модификация COM, обеспечивающая передачу вызовов между компьютерами, называется DCOM (Distributed COM — распределенная COM).

По технологии COM приложение предоставляет свои службы, используя специальные объекты — ***объекты*** COM, которые являются экземплярами ***классов*** COM. Объект COM так же, как обычный объект, включает поля и методы, но в отличие от обычных объектов каждый объект COM может реализовывать несколько интерфейсов, обеспечивающих доступ к его ***полям*** и функциям. Это достигается за счет организации отдельной таблицы адресов методов для каждого интерфейса (по типу таблиц виртуальных методов). При этом интерфейс обычно объединяет несколько однотипных функций. Кроме того, классы COM поддерживают ***наследование интерфейсов,*** но не поддерживают ***наследования реализации,*** т. е. не наследуют код методов, хотя при необходимости объект класса-потомка может вызвать метод родителя (рис. 1.4).

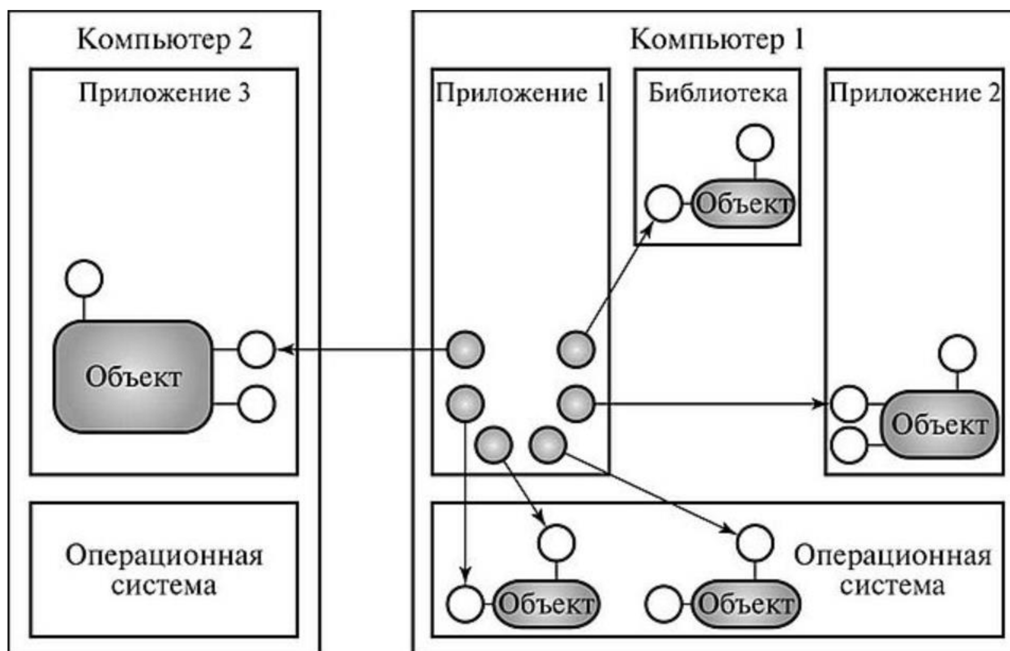

Рис. 1.4 - Взаимодействие программных компонентов различных типов

Каждый интерфейс имеет имя, начинающееся с символа «I», и глобальный уникальный идентификатор HD (Interface IDentifier). Любой объект COM обязательно реализует интерфейс I Unknown (на схемах этот интерфейс всегда располагают сверху). Использование этого интерфейса позволяет получить доступ к остальным интерфейсам объекта.

Объект всегда функционирует в составе ***сервера*** — динамической библиотеки или исполняемого файла, которые обеспечивают функционирование объекта. Различают три типа серверов:

1. внутренний сервер; реализуется динамическими библиотеками, которые подключаются к приложению-клиенту и работают в одном с ними адресном пространстве, наиболее эффективный сервер, кроме того, он не требует специальных средств;
2. локальный сервер; создается отдельным процессом (модулем, exe), который работает на одном компьютере с клиентом;
3. удаленный сервер; создается процессом, который работает на другом компьютере.

Например, Microsoft Word является локальным сервером. Он включает множество объектов, которые могут использоваться другими приложениями.

Для обращения к службам клиент должен получить указатель на соответствующий интерфейс. Перед первым обращением к объекту клиент посылает запрос к библиотеке COM, хранящей информацию обо всех зарегистрированных в системе классах COM объектов, и передает ей имя класса, идентификатор интерфейса и тип сервера. Библиотека запускает необходимый сервер, создает требуемые объекты и возвращает указатели на объекты и интерфейсы. Получив указатели, клиент может вызывать необходимые функции объекта.

Взаимодействие клиента и сервера обеспечивается базовыми механизмами COM или DCOM, поэтому клиенту безразлично местонахождение объекта. При использовании локальных и удаленных серверов в адресном пространстве клиента создается ***proxy-объект*** — заместитель объекта COM, а в адресном пространстве сервера COM — заглушка, соответствующая клиенту. Получив задание от клиента, заместитель упаковывает его параметры и, используя службы операционной системы, передает вызов заглушке. Заглушка распаковывает задание и передает его объекту COM. Результат возвращается клиенту в обратном порядке.

На базе технологии COM и ее распределенной версии DCOM были разработаны компонентные технологии, решающие различные задачи разработки программного обеспечения.

OLE-automation или просто Automation (автоматизация) — технология создания программируемых приложений, обеспечивающая программируемый доступ к внутренним службам этих приложений. Вводит понятие ***диспинтерфейса*** (dispinterface) — специального интерфейса, облегчающего вызов функций объекта. Эту технологию поддерживает, например, Microsoft Excel, предоставляя другим приложениям свои службы.

ActiveX — технология, построенная на базе OLE-automation, предназначена для создания программного обеспечения как сосредоточенного на одном компьютере, так и распределенного в сети. Предполагает использование визуального программирования для создания компонентов — элементов управления ActiveX. Полученные таким образом элементы управления можно устанавливать на компьютер дистанционно с удаленного сервера, причем устанавливаемый код зависит от используемой операционной системы. Это позволяет применять элементы управления ActiveX в клиентских частях приложений Интернета.

Основными преимуществами технологии ActiveX, обеспечивающими ей широкое распространение, являются:

1. быстрое написание программного кода, поскольку все действия, связанные с организацией взаимодействия сервера и клиента, берет на программное обеспечение COM, программирование сетевых приложений становится похожим на программирование для отдельного компьютера;
2. открытость и мобильность — спецификации технологии недавно были переданы в Open Group как основа открытого стандарта;
3. возможность написания приложений с использованием знакомых средств разработки, например, Visual Basic, Visual C++, Borland Delphi, Borland C++ и любых средств разработки на Java;
4. большое количество уже существующих бесплатных программных элементов ActiveX (к тому же практически любой программный компонент OLE совместим с технологиями ActiveX и может применяться без модификаций в сетевых приложениях);
5. стандартность — технология ActiveX основана на широко используемых стандартах Интернет (TCP/IP, HTML, Java), с одной стороны, и стандартах, введенных в свое время Microsoft и необходимых для сохранения совместимости (COM, OLE).

MTS (Microsoft Transaction Server — сервер управления транзакциями) — технология, обеспечивающая безопасность и стабильную работу распределенных приложений при больших объемах передаваемых данных.

MIDAS (Multitier Distributed Application Server — сервер многозвенных распределенных приложений) — технология, организующая доступ к данным разных компьютеров с учетом балансировки нагрузки сети.

Все указанные технологии реализуют компонентный подход, заложенный в COM. Так, с точки зрения COM элемент управления ActiveX — внутренний сервер, поддерживающий технологию OLE-automation. Для программиста же элемент ActiveX — «черный ящик», обладающий свойствами, методами и событиями, который можно использовать как строительный блок при создании приложений.

Технология CORBA, разработанная группой компаний OMC (Object Management Group — группа внедрения объектной технологии программирования), реализует подход, аналогичный COM, на базе объектов и интерфейсов CORBA. Программное ядро CORBA реализовано для всех основных аппаратных и программных платформ, и потому эту технологию можно использовать для создания распределенного программного обеспечения в гетерогенной (разнородной) вычислительной среде. Организация взаимодействия между объектами клиента и сервера в CORBA осуществляется с помощью специального посредника, названного VisiBroker, и другого специализированного программного обеспечения.

Отличительной особенностью современного этапа развития технологии программирования, кроме изменения подхода, является создание и внедрение автоматизированных технологий разработки и сопровождения программного обеспечения, которые были названы CASE-технологиями (Computer-Aided Software / System Engineering — разработка программного обеспечения программных систем с

использованием компьютерной поддержки). Без средств автоматизации разработка достаточно сложного программного обеспечения на настоящий момент становится трудно осуществимой: память человека уже не в состоянии фиксировать все детали, которые необходимо учитывать при разработке программного обеспечения. На сегодня существуют CASE-технологии, поддерживающие как структурный, так и объектный (в том числе и компонентный) подходы к программированию.

Появление нового подхода не означает, что отныне все программное обеспечение будет создаваться из программных компонентов, но анализ существующих проблем разработки сложного программного обеспечения показывает, что он будет применяться достаточно широко.

Вопросы к главе 1

1. Перечислите основные этапы развития разработки программного обеспечения.
2. В какой период времени доминировало стихийное программирование?
3. Как была устроена типичная программа времен стихийного программирования?
4. В чем заключался кризис программирования 60-ых годов?
5. Сформулируйте основные принципы структурного подхода к программированию.
6. Что такое процедурное программирование?
7. Приведите примеры процедурных языков.
8. Что такое модульное программирование?
9. Что такое объектно-ориентированное программирование?
10. Назовите первый объектно-ориентированный язык.

11. Назовите основные преимущества объектно-ориентированного программирования.

12. Что такое инкапсуляция, наследование, полиморфизм?

13. В чем заключается визуальный подход к объектно-ориентированному программированию?

14. Что такое компонентное программирование?

15. Какие технологии обеспечивают реализацию компонентного подхода к программированию?

16. Опишите работу технологии COM.

17. Что такое CASE-технологии?

Глава 2. СТАНДАРТИЗАЦИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Основные понятия в области стандартизации

Стандартизация — это деятельность, направленная на разработку и установление требований, норм, правил, характеристик, как обязательных для выполнения, так и рекомендуемых, обеспечивающая право потребителя на приобретение товаров надлежащего качества, а также право на безопасность и комфортность труда. ***Цель стандартизации*** — достижение оптимальной степени упорядочения в той или иной области посредством широкого и многократного использования установленных положений, требований, норм для решения реально существующих, планируемых или потенциальных задач. ***Основными результатами деятельности по стандартизации*** должны быть повышение степени соответствия продукта (услуги), процессов их функциональному назначению, устранение технических барьеров в международном товарообмене, содействие научно-техническому прогрессу и сотрудничеству в различных областях.

Стандартизация связана с такими понятиями, как объект стандартизации и область стандартизации. Объектом стандартизации обычно называют продукцию, процесс, услугу, для которых разрабатывают те или иные требования, характеристики, параметры, правила и т.п. Стандартизация может касаться либо объекта в целом, либо его отдельных составляющих (характеристик). Областью стандартизации называют совокупность взаимосвязанных объектов стандартизации.

Стандартизация осуществляется на разных уровнях (рис. 2.1). ***Уровень стандартизации*** зависит от того, участники какого географического, экономического, политического региона мира

принимают стандарт. Если участие в стандартизации открыто для соответствующих органов любой страны, то это международная стандартизация. Региональная стандартизация — деятельность, открытая только для соответствующих органов государств одного географического, политического или экономического региона. Региональная и международная стандартизация осуществляется специалистами стран, представленных в соответствующих региональных и международных организациях. ***Национальная стандартизация*** — стандартизация в одном конкретном государстве. При этом национальная стандартизация также может осуществляться на разных уровнях: на государственном, отраслевом, в том или ином секторе экономики (например, на уровне министерств), на уровне ассоциаций, производственных фирм, предприятий (фабрик, заводов) и учреждений.

Стандартизацию, которая проводится в административно-территориальной единице (провинции, крае и т.п.), принято называть ***административно-территориальной стандартизацией***.

Рис. 2.1 - Уровни стандартизации

Стандарт (от англ. standard — норма, образец) — в широком смысле слова образец, эталон, модель, принимаемые за исходные для сопоставления с ними других подобных объектов.

Стандарт как нормативно-технический документ устанавливает комплекс норм, правил, требований к объекту стандартизации. Стандарт может быть разработан как на материальные предметы (продукцию, эталоны, образцы веществ), так и на нормы, правила, требования в различных областях. В переносном смысле — шаблон, трафарет, не содержащий ничего оригинального.

Стандарт — это нормативный документ, разработанный на основе консенсуса, утвержденный признанным органом, направленный на достижение оптимальной степени упорядочения в определенной области. В стандарте устанавливаются для всеобщего и многократного использования общие принципы, правила, характеристики, касающиеся различных видов деятельности или их результатов. Стандарт должен быть основан на обобщенных результатах научных исследований, технических достижений и практического опыта, тогда его использование принесет оптимальную выгоду для общества.

Стандарты имеют большое значение — они обеспечивают возможность разработчикам программного обеспечения использовать данные и программы других разработчиков, осуществлять экспорт/импорт данных.

Стандарты занимают все более значительное место в направлении развития индустрии информационных технологий. Более 250 подкомитетов в официальных организациях по стандартизации работают над стандартами в области информационных технологий. Более 1000 стандартов или уже приняты этими организациями, или находятся в процессе разработки. Процесс стандартизации информационных технологий далеко не закончен (да, по нашему мнению, вряд ли когда-либо будет закончен, так как область информационных технологий постоянно динамично развивается).

Все компании-разработчики должны обеспечить приемлемый уровень качества выпускаемого программного обеспечения (ПО). Для этих целей предназначены стандарты качества программного обеспечения или отдельные разделы в стандартах разработки программного обеспечения, посвященные требованиям к качеству программного обеспечения.

С точки зрения пользователя, все многообразие ПО должно управляться единообразно. Должна быть единообразная навигация — перемещение по программе, единообразные органы управления ПО и единая реакция программного обеспечения на пользовательский

обеспечения наиболее удачно описана во введении в стандарт ISO/ IEC 12207: «Программное обеспечение является неотъемлемой частью информационных технологий и традиционных систем, таких, как транспортные, военные, медицинские и финансовые. Имеется множество разнообразных стандартов, процедур, методов, инструментальных средств и типов операционной среды для разработки и управления программным обеспечением. Это интерфейс — GUI (Graphical User Interface).

Все это регламентируется стандартами, действующими в сфере информационных технологий.

Необходимость стандартизации разработки программного разнообразие создает трудности при проектировании и управлении программным обеспечением, особенно при объединении программных продуктов и сервисных программ. Стратегия разработки программного обеспечения требует перехода от этого множества к общему порядку, который позволит специалистам, практикующимся в программном обеспечении, «говорить на одном языке» при разработке и управлении программным обеспечением. Этот международный стандарт обеспечивает такой общий порядок».

2.2 Стандарты в области управления качеством программного обеспечения

Наиболее популярными стандартами в области качества ПО в настоящее время являются: ISO 9001, TickIT, SEI SW-CMM.

Стандарты ISO серии 9000

Стандарты международной организации по стандартизации ISO считаются самыми известными и распространенными в мире, а также они универсальны, поэтому их можно применять в любой отрасли. Как и у любой другой модели у ISO есть преимущества и недостатки.

Основные преимущества данной модели - это распространенность и признание на мировом уровне. В настоящее время стандарты ISO обязательны для любой существующей на рынке организации.

Но конечно же, вследствие своей универсальности, модель на основе стандартов ISO серии 9000 получилась достаточно «высокоуровневой».

Поэтому для построения полноценной системы качества, основанной на модели ISO, необходимо использовать большое количество вспомогательных отраслевых и ISO стандартов.

Стандарт TickIT

TickIT - британский стандарт, получивший широкую известность. Он регламентирует требования к системе качества для организаций разработчиков программного обеспечения и базируется на модели ISO 9001:94. В отличие от модели ISO 9001, которая регламентирует "что необходимо сделать", разработчики данного стандарта попытались ответить на вопрос "как" можно выполнить требования, определенные в ISO 9001. TickIT объединяет в себе модель ISO 9001 с набором рекомендательных стандартов ISO 12207 и ISO 9000-3.

Стандарты SEI SW-CMM

Основой данной модели является теория TQM (Total Quality Management).

TQM - всеобщий менеджмент качества, появившийся в 60-е годы для обозначения японского подхода к управлению компаниями. Данный подход предполагает непрерывное улучшение качества в различных сферах деятельности, таких, как производство, закупка, сбыт и т.п. Теория TQM основывается на постепенном улучшении внутренних производственных процессов за счет множества небольших внедряемых в компании улучшений. Однако, модели ISO и CMM несколько различаются в своих подходах к построению самосовершенствующихся систем управления качеством и улучшению производственных процессов.

В отличие от модели ISO, где для того, чтобы соответствовать требованиям, необходимо продемонстрировать 100%-ное соответствие модели (и только оно позволяет компании самосовершенствоваться), в модели SEI SW-CMM предусмотрен поэтапный подход к построению системы совершенствования процессов. Для достижения этой цели разработчики стандарта CMM определили пять уровней, которые должна пройти организация для того, чтобы достичь основной цели – повышения эффективности функционирования процессов компании и, как следствие, улучшения качества результатов производственных процессов и разрабатываемого программного обеспечения.

2.3 Стандарты в области управления проектами разработки программного обеспечения

Одним из важных моментов, который необходимо иметь в виду при внедрении каких-либо стандартов (ISO 9000, SEI SW-CMM, TickIT, Spice ISO 15504 и т.п.), связан с тем, что структура

производства компаний, разрабатывающих программное обеспечение, связана со спецификой продукта. Каждый продукт, разрабатываемый IT-компанией, уникален. И для его разработки, как правило, используется проектный тип организации производства, который тесно связан с матричной структурой управления проектами.

Управление проектами – это приложение знаний, опыта, методов и средств к работам проекта для удовлетворения требований, предъявляемых к проекту, и ожиданий участников проекта. Чтобы удовлетворить эти требования и ожидания, необходимо найти оптимальное сочетание между целями, сроками, затратами, качеством и другими характеристиками проекта.

176 комитет ISO разработал рекомендательный стандарт ISO 10006 "Менеджмент качества. Руководство качеством при управлении проектами", который определяет основные подходы к управлению проектами и определяет его место в модели обеспечения качеством. Авторы стандартов ISO серии 9000 определяют процесс управления проектами как часть системы менеджмента качества. С другой стороны, возможен и противоположный взгляд (которого придерживаются оппоненты стандартов ISO серии 9000), согласно которому менеджмент качества является одной из составных частей системы управления проектами.

Управление проектами является скелетом производства в организациях разработчиков программного обеспечения. Поэтому неудивительно, что для приведения в соответствие системы управления качеством производства к требованиям модели ISO 9001 и к требованиям модели улучшения процессов производства SEI SW-CMM использование стандартов и признанных в мире технологий по управлению проектами является краеугольным камнем развития внутренних технологий в IT-компаниях.

2.4 Стандарты в области проектирования программного обеспечения

ISO / IEC 12207 - базовый стандарт на процессы жизненного цикла ИС, ориентированный на разные типы проектов. В стандарте не предусмотрено конкретных этапов жизненного цикла ИС. Вместо этого был определен только ряд процессов. Поэтому стандарт позволяет реализовать произвольную модель жизненного цикла, и это является его достоинством.

Стандарт ISO / IEC 12207 - важнейший нормативный документ, регламентирующий жизненный цикл программного обеспечения. Он определяет структуру жизненного цикла, содержащие действия, задачи и процессы, которые обязаны быть выполнены в период создания ПО. Подобные регламенты стали общими для любых моделей жизненного цикла, технологий и методологий разработка ПО. Способы выполнения действий и задач, включенных в перечисленных процессах, могут быть произвольного типа.

Согласно с международным стандартом ISO / IEC 12207 процессы жизненного цикла делятся на 3 группы:

1. Главные процессы:
   a. приобретение - устанавливает действия предприятия-покупателя;
   b. поставки - устанавливает действия предприятия-поставщика;
   c. разработка - устанавливает действия предприятия-разработчика;
   d. функционирование - устанавливает действия предприятия - оператора, обеспечивающего обслуживание системы в целом в процессе ее функционирования;
   e. сопровождение – устанавливает действия персонала, обеспечивающего сопровождение программы, т. е.

управление, модификацию, поддержку функциональной пригодности и текущего состояния;

f. инсталляция ПО на вычислительной системе и его удаления.

2. Вспомогательные процессы - процессы, предназначенные для поддержки главных составляющих, организации верификации, качества проекта, тестирования и проверки ПО и т.д.:

a) процесс документирования;
b) процесс обеспечения качества;
c) процесс управления конфигурацией;
d) процесс аттестации;
e) процесс верификации;
f) процесс аудита;
g) процесс решения проблем;
h) процесс совместной оценки.

3. Организационные процессы определяют задачи и действия, которые выполняет как заказчик, так и разработчик проекта, а именно:

a) процесс создания инфраструктуры проекта;
b) процесс управления;
c) процесс обучения;
d) процесс усовершенствования.

ГОСТ Р ИСО/МЭК 12207-99 «Информационная технология. Процессы жизненного цикла программных средств». «Программное обеспечение - это неотъемлемая часть информационных технологий и традиционных систем, например, транспортные, военные, медицинские и финансовые. Имеется множество разнообразных стандартов, процедур, методов, инструментальных средств и типов операционной среды для разработки и управления программным обеспечением. Это разнообразие создает трудности при проектировании и управлении программным обеспечением, особенно

при объединении программных продуктов и сервисных программ. Стратегия разработки программного обеспечения требует перехода от этого множества к общему порядку, который позволит специалистам, практикующимся в программном обеспечении, «говорить на одном языке» при разработке и управлении программным обеспечением. Этот международный стандарт обеспечивает такой общий порядок».

Стандарт ГОСТ Р ИСО/МЭК 12207-99 определяет базовое понятие программной системы – «жизненный цикл» (ГОСТ Р ИСО/МЭК ТО 15271-2002 «Информационная технология. Руководство по применению ГОСТ Р ИСО/МЭК 12207»).

ГОСТ Р ИСО/МЭК 12207-99 вводит понятие модели жизненного цикла как структуры, состоящей из процессов, и охватывающей жизнь системы от установления требований к ней до прекращения ее использования. Предлагается это определение подкорректировать и разделить на два определения:

1. жизненный цикл – совокупность процессов, разделенных на работы и задачи, и включающих в себя разработку, эксплуатацию и сопровождение программного продукта, охватывающих жизнь системы от установления требований к ней до прекращения ее использования.

2. модель жизненного цикла – структура, определяющая последовательность осуществления процессов, работ и задач, выполняемых на протяжении жизненного цикла программной системы, а также взаимосвязи между ними.

ГОСТ 34.601-90 - распространяется на автоматизированные информационные системы и регламентирует стадии, этапы их создания, содержит описание содержания работ на каждом из этапов. Стандарт ориентирован на использование каскадной модели жизненного цикла.

- формирование требований к АС,
- разработка концепции АС,

- техническое задание,
- эскизный проект,
- технический проект,
- рабочая документация,
- ввод в действие,
- сопровождение.

ГОСТ Р ИСО/МЭК 12119-2000 «Информационная технология. Пакеты программ. Требования к качеству и тестирование» содержит указания, определяющие порядок тестирования продукта на соответствие его требованиям к качеству. Тестирование - это трудоемкий процесс. Согласно оценкам некоторых специалистов процентное распределение времени между процессами проектирование – разработка – тестирование находится в отношении 40-20-40. В этой связи широкое распространение получают системы автоматизации тестирования. В стандарте IEEE 1209-1992 «Recommended Practice for the Evaluation and Selection of CASE Tools» сформулированы общие требования к функциям средств автоматизации тестирования.

Интеграция системы заключается в сборке всех ее компонентов, включая ПС и оборудование. После интеграции система, в свою очередь, подвергается квалификационному тестированию на соответствие совокупности требований к ней. При этом также производятся оформление и проверка полного комплекта документации на систему.

Установка системы осуществляется разработчиком в соответствии с планом в той среде и на том оборудовании, которые предусмотрены договором. В процессе установки проверяется работоспособность программного обеспечения.

Приемка системы - это оценка результатов квалификационного тестирования ПС и системы и документирование результатов оценки, которые проводятся заказчиком с помощью разработчика.

Разработчик выполняет окончательную передачу ПС заказчику в соответствии с договором, обеспечивая при этом необходимое обучение и поддержку.

Вопросы к главе 2

1. Что такое стандартизация?
2. Что такое стандарт?
3. Какие бывают уровни стандартизации?
4. Какие бывают стандарты в области разработки программного обеспечения?
5. Какие Вы знаете организации, занимающиеся стандартизацией разработки программного обеспечения?
6. Приведите примеры стандартов в области качества программного обеспечения?
7. Что такое управление проектами?
8. Приведите примеры стандартов в области управления проектами разработки программного обеспечения?
9. Перечислите группы процессов разработки программного обеспечения в соответствии с ISO 12207.
10. Перечислите основные процессы разработки программного обеспечения в соответствии с ISO 12207.
11. Перечислите вспомогательные процессы разработки программного обеспечения в соответствии с ISO 12207.
12. Перечислите организационные процессы разработки программного обеспечения в соответствии с ISO 12207.
13. Что такое жизненный цикл программного средства?
14. Приведите примеры стандартов в области тестирования программного обеспечения.

Глава 3. МЕТОДОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

3.1 Понятие методологии разработки ПО

Методология разработки ПО — организация труда, включающая идеологические принципы, план, контроль над процессами, подход к сотрудникам.

В современном мире существует множество моделей разработки ПО. Сегодня принята стандартная классификация:

Классические:

- каскадная;
- итерационная;
- спиральная.

Гибкие:

- agile;
- scrum;
- XP;
- и т.д.

Существует также ряд методологий, ориентированных на использование специализированных CASE-технологий, например, RAD и RUP.

Далее мы кратко рассмотрим ряд наиболее популярных методологий в области разработки программного обеспечения.

3.2 Модель Waterfall

Модель Waterfall также называется каскадной, водопадной или последовательной. Данная модель является старейшей получившей широкую известность моделью процесса разработки программного обеспечения, с помощью которой действительно можно

структурировать процесс разработки. Каскадная модель была предложена Уинстоном Рейсом в 1970 году.

Водопадная модель относится к классическому пониманию разработки ПО. Весь процесс является жестким и линейным, имеет четкие цели для каждого этапа. Наиболее распространенный список этапов разработки ПО в соответствии с каскадной моделью показан на рис. 3.1. В ГОСТ 12207-99 в ГОСТ 34.601-90 «Автоматизированные системы. Стадии создания» этапы немного различаются по составу.

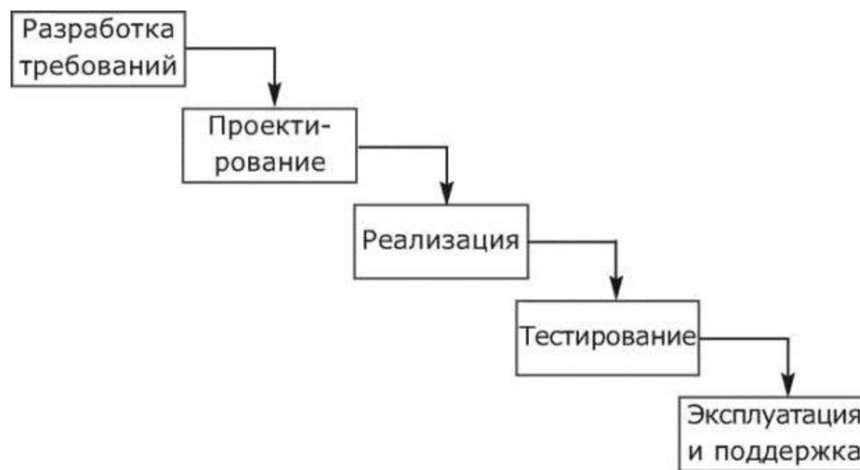

Рис. 3.1 – Водопадная (каскадная) модель ЖЦ разработки

Принципы работы

Основным постулатом данной модели является то, что следующий этап не может быть начат, пока не закончен предыдущий. Так как каждая стадия водопадной модели заканчивается получением некоторых результатов, которые нужны в качестве исходных данных для следующей стадии. При этом не разрешаются произвольные переходы от одного этапа к другому. Процессы осуществляются строго последовательно.

Рассмотрим более подробно действия, выполняемые на каждом из этапов.

Разработка требований

На этом этапе важно задокументировать все требования к будущему программному обеспечению. Необходимо посвятить достаточно времени обсуждению деталей проекта со всеми заинтересованными сторонами. Все поступающие данные нужно проанализировать и систематизировать. Важно также учесть все технические ограничения, которые могут возникнуть на стороне заказчика. Итогом данного этапа должно стать создание подробной спецификации, отвечающей всем требованиям заказчика. Также следует обратить внимание и на другие факторы, которые могут затруднять процесс разработки. К ним относятся дедлайны, установленные заказчиком, а также бюджетные ограничения.

Обратите внимание: чем больше информации о проекте вы соберете, тем меньше времени потратите на исправление ошибок, доработку проекта, пересмотры бюджета, обсуждения и решение других вопросов.

Важной задачей является создание подробного ***документа видения (или образа) проекта***, который включает краткое описание проекта, бизнес-цели, а также критерии успеха проекта, факторы бизнес-рисков и описание конечного пользователя продукта.

Готовый документ необходимо передать на утверждение заказчику, чтобы убедиться в том, что все поставленные требования были учтены, а также чтобы проинформировать его о любых рисках, которые могут возникнуть после релиза проекта.

После того, как все основные вопросы решены, рекомендуется провести дополнительные обсуждения и интерактивные семинары со всеми заинтересованными сторонами. Это поможет выявить какие-либо неочевидные моменты, которые в дальнейшем могут стать причиной внесения изменений в интерфейс приложения или необходимости переписывания паттернов кода. Данный этап может

также включать заполнение анкет, рассмотрение кейсов, мозговой штурм и т.д.

Многие проекты заходят в тупик из-за дополнительных требований, которые всплывают на стадии разработки. Поэтому очень важно понимать начальные бизнес-цели и главную идею будущего приложения.

Спецификация требований программного обеспечения (SRS) описывает требования, которым должно отвечать создаваемое программное обеспечение. Она должна быть логичной, последовательной, доступной и полной. Требования могут выражаться в разных формах, например, в виде традиционных утверждений долженствования (н-р, «Система Staff Manager должна поддерживать следующие браузеры: Google Chrome, Apple Safari, Mozilla Firefox, Opera, IE 8+») или в виде пользовательских историй (н-р, «поскольку я являюсь менеджером, мне необходим доступ к персональной информации всех сотрудников»).

Существует большое количество шаблонов спецификаций. Выбор определенного шаблона зависит от специфики проекта. В большинстве случаев, спецификация включает в себя описание продукта, классы пользователей, функциональные и нефункциональные требования к разрабатываемому программному обеспечению. Иногда в шаблон также входит прототип. Главное — сделать спецификацию понятной, лаконичной и полезной для разработчиков.

Для создания прототипа вам необходимо выяснить следующее:

- способ получения и обработки входящих данных для создания необходимых данных на выходе;
- форма, в которой должны быть представлены выходные данные.

Мокапы (или прототипы) передаются UI/UX-дизайнерам, которые превращают их в красочные шаблоны.

Проектирование

Следующим этапом жизненного цикла ПО является создание документа, описывающего масштабы и границы проекта. Данный документ включает в себя мокапы или скетчи интерфейса будущего приложения, а также подробную спецификацию требований программного обеспечения. Необходимо отметить, что в некоторых случаях документ видения (образа) проекта и документ о масштабах и границах проекта могут быть представлены как единый документ «Об образе и границах проекта».

В документе, описывающем масштабы и границы проекта, должны быть перечислены основные функции создаваемого программного обеспечения. Они определяются на основании документа видения проекта, и безусловно, с учетом указанных временных рамок и установленного бюджета. Кроме того, в данный документ входят мокапы или скетчи, созданные на основе документа видения проекта, а также собранных требований. Вы можете нарисовать скетч пользовательского интерфейса от руки либо использовать для этого программы создания мокапов, и затем согласовать его с заказчиком.

В процессе обсуждения проекта у заказчика может появляться все больше новых идей относительно его реализации. Поэтому рекомендуется дать ему время на обдумывание своего проекта и требований к нему, а затем повторно собраться и обсудить детали проекта, чтобы ничего не упустить из вида.

На этом этапе также поднимается вопрос о послепродажном обслуживании продукта. Вы должны уведомить заказчика о том, каким образом будет осуществляться техническая поддержка после завершения этапа тестирования и последующего релиза продукта.

Обратите внимание на то, что документ видения проекта и документ о масштабах и границах проекта должны быть созданы до подписания контракта.

Разработка

Необходимо отметить, что разработка программного обеспечения может также включать в себя создание интерактивного прототипа, который, в сущности, является основой будущего приложения. Такой прототип помогает определить архитектуру системы в целом. На данном этапе пишется мало кода: например, код кнопок и простых форм, чтобы дать заказчику общее представление о том, как будет работать конечный продукт. Поэтому мы включили создание прототипа в этап разработки программного обеспечения.

Как только интерактивный прототип и дизайн приложения готов и утвержден заказчиком, начинается разработка стандартов приложения (конвенции наименований, способа документирования кода, инструкций для конечного пользователя и т.д.). После этого можно смело переходить к следующему этапу жизненного цикла, а именно, к разработке программного обеспечения. Разработка ПО может быть разделена на небольшие части, или юниты, и каждый юнит разрабатывается и тестируется разработчиками для проверки его функциональности (модульное тестирование).

Тестирование

После завершения этапа разработки продукт должен пройти тщательное тестирование, чтобы убедиться в том, что он соответствует поставленным требованиям. На этапе приемочного тестирования необходимо, чтобы заказчик попытался применить продукт локально точно таким же образом, как он собирается использовать его после релиза. Когда будут исправлены основные ошибки, программное обеспечение можно внедрять. Для исправления незначительных ошибок может использоваться простая система отслеживания, что позволит исправлять любые недоработки уже на этапе сопровождения ПО.

Эксплуатация и поддержка

После того, как продукт был протестирован и развернут на сервере заказчика, начинается следующая фаза жизненного цикла разработки программного обеспечения, которая называется сопровождением или технической поддержкой ПО. В целом, сопровождение подразумевает под собой исправление мелких багов, которые обнаруживаются на этом этапе. Тем не менее, вполне возможно, что вам придется вносить некоторые изменения в созданное программное обеспечение, несмотря на все усилия, приложенные вами на предыдущих этапах. Заказчик может решить внести изменения в функциональность разработанного продукта. Следовательно, вам придется собирать, описывать и обсуждать новые требования с заказчиком, чтобы внести в продукт необходимые изменения. В данном случае, вам предстоит работа с новым каскадным проектом, и все вышеописанные шаги придется повторять с начала.

Преимущества модели

1. Предельная детализация каждого этапа работы, сопровождающаяся документированием.

2. Требования максимально внятно и четко изложены, не могут меняться в процессе работы.

3. Возможность заранее знать сколько денежных и временных ресурсов будет затрачено на проект.

4. Легкость понимания методологии.

5. Простота контроля выполнения задачи.

6. При необходимости легко передать проект другой команде.

Недостатки модели

1. Главный недостаток каскадной модели заключается в том, что ошибки и недоработки на любом из этапов проявляются, как правило, на последующих этапах работ, что приводит к необходимости возврата назад.

2. Дополнительные временные затраты на ведение документации.

3. Необходимость дополнительных квалифицированных кадров для создания технического задания.

4. Высокие временные и денежные затраты.

5. Позднее тестирование.

6. Сложность распараллеливания работ.

7. Сложность управления проектом.

8. Высокий уровень риска и ненадежность инвестиций (возврат на предыдущие стадии может быть связан не только с ошибками, но и с изменениями, произошедшими в предметной области или в требованиях заказчика во время разработки. Причем возврат проекта на доработку вследствие этих причин не гарантирует, что предметная область снова не изменится к тому моменту, когда будет готова следующая версия проекта. Фактически это означает, что существует вероятность того, что процесс разработки «зациклится» и система никогда не дойдет до сдачи в эксплуатацию. Расходы на проект будут постоянно расти, а сроки сдачи готового продукта постоянно откладываться).

Каскадная модель демонстрирует классический подход к разработке различных систем в различных прикладных областях. Для разработки информационных систем данная модель широко использовалась в 70-х и первой половине 80-х годов. Каскадная модель предусматривает последовательную организацию процессов. Причем переход к следующему процессу происходит только после того, как полностью завершены все работы на предыдущем. Каждый процесс завершается выпуском полного комплекта документации, достаточной для того, чтобы работа могла быть продолжена другой командой разработчиков.

На практике водопадная модель часто не оправдывает ожиданий, поскольку игнорирует динамические изменения. Так,

после тестирования очень сложно откатить процесс и заложить функции, не учтенные на стадии разработки. Каскадная модель неэффективна ещё и потому, что предполагает временные простои сотрудников в рамках одного проекта. Тестирование проводится только в конце разработки, хотя проблемы, найденные на этом этапе — это дорогостоящие исправления.

3.3 Итерационная модель

Рис. 3.2 - Итерационная модель

Не все модели жизненного цикла последовательны. Существуют также итеративные (или инкрементальные) модели, в которых используется другой подход. Вместо одной продолжительной последовательности действий здесь весь жизненный цикл продукта разбит на ряд отдельных мини-циклов (рис. 3.2). Причем каждый из них состоит из все тех же базовых стадий модели жизненного цикла. Эти мини-циклы называются итерациями. В каждой из итераций происходит разработка отдельного компонента системы, после чего этот компонент добавляется к уже ранее разработанному функционалу.

Итеративная модель не предполагает полного объема требований для начала работ над продуктом. Разработка программы может начинаться с требований к части функционала, которые могут впоследствии дополняться и изменяться. Процесс повторяется, обеспечивая создание новой версии продукта для каждого цикла.

В несколько упрощенном виде, итеративная модель состоит из четырех основных стадий, которые повторяются в каждой из итераций (plan-do-check-act):

– определение и анализ требований;

– дизайн и проектирование – согласно требованиями. Причем дизайн может как разрабатываться отдельно для данной функциональности, так и дополнять уже существующий;

– разработка и тестирование – кодирование, интеграция и тестирование нового компонента;

– фаза ревью – оценка, пересмотр текущих требований и предложения дополнений к ним.

По результатам каждой итерации принимается решение – будут ли использованы ее результаты для дополнения существующей функциональности в качестве входной точки для начала следующей итерации (т.н. инкрементальное прототипирование). В конечном итоге, достигается точка, в которой все требования были воплощены в продукте – происходит релиз.

В математических терминах, итеративная модель представляет реализацию методики последовательной аппроксимации – то есть, постепенное приближение к образу готового продукта.

Ключ к успешному использованию этой модели – строгая валидация требований и тщательная верификация разрабатываемой функциональности в каждой из итераций.

Основные стадии процесса разработки в итеративной модели фактически повторяют модель водопада. В каждой итерации

создается программное обеспечение, требующее тестирования на всех уровнях.

Плюсы и минусы итеративной модели:

+ раннее создание работающего ПО;

+ гибкость – готовность к изменению требований на любом этапе разработки;

+ каждая итерация – маленький этап, для которого тестирование и анализ рисков обеспечить проще, чем для всего жизненного цикла продукта.

— каждая фаза – самостоятельна, отдельные итерации не накладываются;

— могут возникнуть проблемы с реализацией общей архитектуры системы, поскольку не все требования известны к началу проектирования.

Когда использовать итеративную модель:

— для крупных проектов;

— когда известны, по крайней мере, ключевые требования;

— когда требования к проекту могут меняться в процессе разработки.

3.4 Спиральная модель

Спиральная модель (рис. 3.3), которая была предложена Барри Боэмом в 1986 году, стала прорывом в разработке ПО, так как в ней сочетаются итеративность и этапность.

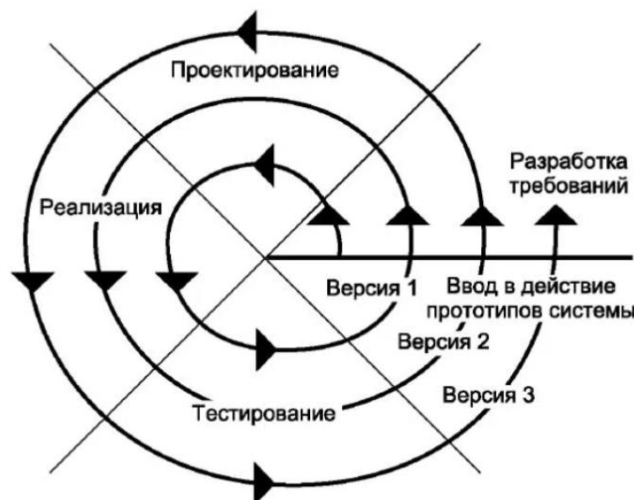

Рис. 3.3 – Схема работы спиральной модели

Модель спирального жизненного цикла — это сложная организация жизненного цикла ПО, которая фокусируется на раннем выявлении и уменьшении проектных рисков.

Боэм сформулирован десять наиболее распространенных рисков:

1. Дефицит специалистов.
2. Нереалистичные сроки и бюджет.
3. Реализация не соответствует функциональности.
4. Разработка неправильного пользовательского интерфейса.
5. Ненужная оптимизация.
6. Непрекращающийся поток изменений.
7. Нехватка информации о внешних компонентах, определяющих окружение системы или вовлечённых в интеграцию.
8. Недостатки в работах, выполняемых внешними (по отношению к проекту) ресурсами.
9. Недостаточная производительность получаемой системы.
10. Разрыв между квалификацией специалистов и требованиями проекта.

Разработка начинается в небольшом масштабе, решаются локальные задачи, оцениваются риски и пути их уменьшения.

Следующий шаг охватывает более комплексные задачи — следующий виток спирали.

Каждый виток разбит на 4 сектора:

1. Определение целей.
2. Оценка и разрешение рисков.
3. Разработка и тестирование.
4. Планирование следующей итерации.

На каждом витке спирали могут применяться разные модели процесса разработки ПО. В конечном итоге на выходе получается готовый продукт.

Главной задачей данной методологии является как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Основной проблема спирального цикла является определение момента перехода на следующий этап. Для её решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла.

Положительные стороны спиральной модели:

1. Улучшенный анализ рисков;
2. хорошая документация процесса разработки;
3. гибкость – возможность внесения изменений и добавления новой функциональности даже на относительно поздних этапах;
4. раннее создание рабочих прототипов.

Отрицательные стороны:

1. достаточно высокая цена использования;
2. привлечение высококлассных специалистов;
3. успех процесса в основном зависит от стадии анализа рисков.

Спиральную модель нужно использовать, когда важен анализ рисков и затрат; есть крупные долгосрочные проекты без четких требований и при разработке новой линейки продуктов.

3.5 Agile

Agile — метод гибкой разработки программного обеспечения, предполагающий большое количество итераций. Основным документом данной методологии является «Манифест о гибкой разработке программного обеспечения Agile».

Манифест Agile включает в себя 4 базовых идеи и 12 принципов эффективного управления проектами.

Идеи Agile:

1. Люди и их взаимодействие важнее, чем процессы и инструменты.
2. Рабочее ПО важнее, чем документация.
3. Клиенты и сотрудничество с ними важнее, чем контракт и обсуждение условий.
4. Готовность к внесению изменений важнее, чем первоначальный план.

Принципы Agile:

1. Удовлетворять клиентов, заблаговременно и постоянно поставляя ПО.
2. Изменять требования к конечному продукту в течение всего цикла его разработки.
3. Поставлять рабочее ПО как можно чаще.
4. Поддерживать сотрудничество между разработчиками и заказчиком в течение всего цикла разработки.
5. Поддерживать и мотивировать всех, кто вовлечен в проект (если команда мотивирована, то она лучше справляется со своими задачами).
6. Обеспечивать непосредственное взаимодействие между разработчиками.
7. Измерять прогресс только посредством рабочего ПО (клиенты должны получать только функциональное и рабочее ПО).

8. Поддерживать непрерывный темп работы.

9. Уделять внимание дизайну и техническим деталям.

10. Стараться сделать рабочий процесс максимально простым, а ПО – простым и понятным.

11. Позволять членам команды самостоятельно принимать решения (если разработчики могут сами принимать решения, самоорганизовываться и общаться с другими членами коллектива, обмениваясь с ними идеями, вероятность создания качественного продукта существенно возрастает).

12. Постоянно адаптироваться к меняющейся среде (благодаря этому конечный продукт будет более конкурентоспособен).

Как и у любой другой методологии, у Agile есть свои положительные и негативные стороны (рис. 3.4). Начнем с положительных:

1. При использовании методологии Agile происходит гибкое управление проектом, благодаря этому учитывается больше требований заказчика и потребителей.

2. Минимизация дефектов конечного продукта, так как происходит тщательная проверка качества после каждого спринта.

3. Agile быстро запускается и легко реагирует на изменения.

4. Возможность команде разработчиков и клиентов поддерживать постоянную связь в реальном времени.

Рис. 3.4 – Разработка по методологии Agile

Также не стоит забывать и о недостатках:

1. Постоянная обратная связь может приводить к тому, что все время будет переноситься.

2. Адаптация под изменяющиеся условия проекта проектную документацию, так как при ненадлежащем информировании команды документы с функциональными требованиями или архитектурой могут оказаться неактуальными на текущий момент времени.

3. Необходимость в частых встречах, так как происходит постоянное отвлечение членов команды от решаемых задач.

4. Необходимость в постоянном присутствии клиента.

5. Невозможность выстраивать долгосрочные планы.

6. Потребность в мотивированных и высококвалифицированных специалистах.

3.6 Методология Scrum

Главное отличие Scrum от других методов системы Agile - основной упор делается на качественный контроль рабочего процесса. Scrum заключается в том, что разработка проекта разделяется на спринты, по окончании которых клиент получает улучшенное ПО. Спринты строго фиксируются по времени и могут длиться от 2 до 4 недель. Рабочий процесс в одном спринте содержит несколько стадий (рис. 3.5):

1. Определение объемов работ.

2. Каждый день проводятся 15-минутные встречи, для того, чтобы каждый член команды мог скорректировать свою работу и подвести итоги на данный промежуток времени.

3. Демонстрация полученных результатов.

Обсуждение спринтов нужно для поиска удачных и неудачных решений и действий.

Чаще всего Scrum используется в работе со сложным программным обеспечением и для разработки продукта с использованием инкрементных и итеративных методов. С помощью этого серьезно повышается производительность команды и сокращаются временные затраты на достижение цели.

Scrum улучшает результаты, помогает адаптировать проект к изменениям, обеспечивает более точную оценку при меньших трудозатратах на анализ и позволяет эффективнее контролировать этапы работы и сценарий проекта.

Рис. 3.5 – Разработка с использованием метода Scrum

3.7 Методология XP

Рис. 3.6 – Методология XP

Экстремальное программирование (XP) — одна из гибких методологий разработки программного обеспечения. Суть методологии - возможность вести разработку в условиях постоянно меняющихся требований (рис. 3.6).

Основные признаки XP:

1. Игра в планирование

Основная цель игры в планирование — быстро сформировать приблизительный план работы и постоянно обновлять его по мере того, как условия задачи становятся всё более чёткими. Артефактами игры в планирование является набор бумажных карточек, на которых записаны пожелания заказчика (customer stories), и приблизительный план работы по выпуску следующих одной или нескольких небольших версий продукта. Критическим фактором, благодаря которому такой стиль планирования оказывается эффективным, является то, что в данном случае заказчик отвечает за принятие бизнес-решений, а команда разработчиков отвечает за принятие технических решений. Если не выполняется это правило, весь процесс распадается на части.

2. План релизов

План релизов определяет даты релизов и формулировки пользователей, которые будут воплощены в каждом из них. Исходя из этого можно выбрать формулировки для очередной итерации. В течение итерации изготавливаются тесты приемки, которые выполняются в пределах этой итерации и всех последующих, чтобы обеспечить правильную работу программы. План может быть пересмотрен в случае значительного отставания или опережения по итогам одной из итераций.

Итерации. Итерации придают процессу разработки динамичность. Не нужно планировать ваши программные задачи надолго вперед. Лучше вместо этого устраивать совещание для планирования в начале каждой итерации. Не стоит и пытаться реализовать то, что не было запланировано. У вас еще будет время, чтобы реализовать эти идеи, когда до них дойдет очередь согласно плану релизов.

Привыкнув, не добавлять функциональность заранее и используя непосредственное планирование, вы сможете легко приспосабливаться к изменчивым требованиям заказчика.

3. Планирование итераций

Планирование итераций начинается со встречи в начале каждой итерации с целью выработки плана шагов для решения программных задач. Каждая итерация должна длиться от одной до трех недель. Формулировки внутри итерации сортируются в порядке их значимости для заказчика. Кроме того, добавляются задачи, которые не смогли пройти тесты приемки и требуют доработки. Формулировки и результаты тестов переводятся в программные задачи. Задачи записываются на карточках, которые образуют детальный план итерации. Для решения к каждой из задач требуется от одного до трех дней. Задачи, для которых нужно менее одного дня, можно сгруппировать вместе, а большие задачи разделить на

несколько мелких. Разработчики оценивают задачи и сроки, для их выполнения.

Для разработчика очень важно точно установить время выполнения задачи. Возможно, потребуется переоценить некоторые формулировки и пересмотреть план релиза после каждых трех или пяти итераций — это вполне допустимо. Если вы в первую очередь реализуете наиболее важные участки работы, то вы всегда будете успевать сделать максимум возможного для ваших клиентов. Стиль разработки, основанный на последовательности итераций, улучшает процесс разработки.

4. Собрание стоя

Каждое утро проводится собрание для обсуждения проблем, их решений и для усиления концентрации команды. Собрание проводится стоя во избежание длительных дискуссий неинтересных всем членам команды. Большое количество времени людей тратится чтобы получить небольшое количество коммуникации. Поэтому участие всех людей в собраниях уводит ресурсы из проекта и создает хаос в планировании.

Для такого рода коммуникаций и нужно собрание стоя. Намного лучше иметь одно короткое обязательное собрание, чем множество длинных на которых большинство разработчиков должно все равно присутствовать.

Ежедневное утреннее собрание позволит избежать многих других собраний и сэкономит больше времени, чем на него затрачено.

5. Простота

Простой дизайн всегда занимает меньше времени, чем сложный. Поэтому всегда делайте самые простые вещи, которые только смогут работать. Всегда быстрее и дешевле заменить сложный код сразу, прежде чем вы потратите много времени на работу с ним. Сохраняйте вещи такими простыми, как только возможно, не добавляя

функциональность до того, как это запланировано. Имейте в виду: сохранять дизайн простым — это тяжелая работа.

6. Система метафор

Выбор системы метафор нужен для удержания команды в одних и тех же рамках при именовании классов и методов. Название объектов важно для понимания общего дизайна системы и повторного использования кодов. Если разработчик в состоянии правильно предугадать, как может быть назван существующий объект, это ведет к экономии времени.

7. Заказчик на рабочей площадке

Основная проблема разработки программного обеспечения - недостаток знаний программистов в разрабатываемой предметной области. Для решения этой проблемы используется участие заказчика в процессе разработки.

Экстремальное программирование учит нас находить самые простые решения — будет задан заказчику прямой вопрос. Более строгие подходы требуют всеобъемлющего предварительного анализа разрабатываемой области. Реальный опыт ведения приземленных проектов показывает, что невозможно собрать все требования заранее. Для этого фиксируется User Story — это описание того как система должна работать. Каждая User Story написана на карточке и представляет какой-то кусок функциональности системы, имеющий логический смысл с точки зрения Заказчика.

8. Тестирование до начала разработки

XP предполагает написание автоматических тестов (программный код, написанный специально для того, чтобы тестировать логику другого программного кода). Особое внимание уделяется двум разновидностям тестирования:

- юнит-тестирование модулей;
- функциональное тестирование.

Разработчик не может быть уверен в правильности написанного им кода до тех пор, пока не сработают абсолютно все тесты модулей разрабатываемой им системы. Тесты модулей (юнит-тесты) позволяют разработчикам убедиться в том, что каждый из них по отдельности работает корректно. Они также помогают другим разработчикам понять, зачем нужен тот или иной фрагмент кода, и как он функционирует — в ходе изучения кода тестов логика работы тестируемого кода становится понятной, так как видно, как он должен использоваться. Тесты модулей также позволяют разработчику без каких-либо опасений выполнять рефакторинг.

Функциональные тесты предназначены для тестирования функционирования логики, образуемой взаимодействием нескольких (часто — довольно внушительного размера) частей. Они менее детальны, чем юнит-тесты, но покрывают гораздо больше — то есть, у тестов, которые при своём выполнении затрагивают больший объём кода, шанс обнаружить какое-либо некорректное поведение, очевидно, больше. По этой причине в промышленном программировании написание функциональных тестов нередко имеет больший приоритет, чем написание юнит-тестов.

Для XP более приоритетным является подход, называемый TDD (разработка через тестирование). В соответствии с этим подходом сначала пишется тест, который изначально не проходит (так как логики, которую он должен проверять, ещё просто не существует), затем реализуется логика, необходимая для того, чтобы тест прошёл. TDD, в некотором смысле, позволяет писать код, более удобный в использовании — потому что при написании теста, когда логики ещё нет, проще всего позаботиться об удобстве будущей системы.

9. Парное программирование

Парное программирование предполагает, что весь код создается парами программистов, работающих за одним компьютером. Один из них работает непосредственно с текстом программы, другой

просматривает его работу и следит за общей картиной происходящего. При необходимости клавиатура свободно передаётся от одного к другому. В течение работы над проектом пары не фиксированы: рекомендуется их перемешивать, чтобы каждый программист в команде имел хорошее представление обо всей системе. Таким образом, парное программирование усиливает взаимодействие внутри команды.

10. Смена позиций

Во время очередной итерации всех работников следует перемещать на новые участки работы. Подобные перемещения необходимы, чтобы избежать изоляции знаний и устранить «узкие места». Особенно плодотворной является замена одного из разработчиков при парном программировании.

11. Коллективное владение кодом

Коллективное владение кодом стимулирует разработчиков подавать идеи для всех частей проекта, а не только для своих модулей. Любой разработчик может изменять любой код для расширения функциональности и исправления ошибок.

С первого взгляда это выглядит как хаос. Однако, принимая во внимание, что как минимум любой код создан парой разработчиков, что тесты позволяют проверить корректность внесенных изменений и что в реальной жизни все равно так или иначе приходится разбираться в чужом коде, становится ясно, что коллективное владение кодом значительно упрощает внесение изменений и снижает риск, связанный с высокой специализацией того или иного члена команды.

12. Соглашение о кодировании

Вы в команде, которая работает над данным проектом продолжительное время. Люди приходят и уходят. Никто не кодирует в одиночку и код принадлежит всем. Всегда будут моменты, когда необходимо будет понять и скорректировать чужой код. Разработчики

будут удалять или изменять дублирующий код, анализировать и улучшать чужие классы и т.п. Со временем нельзя будет сказать кто автор конкретного класса.

Следовательно, все должны подчиняться общим стандартам кодирования — форматирование кода, именование классов, переменных, констант, стиль комментариев. Таким образом, мы будем уверены, что, внося изменения в чужой код (что необходимо для агрессивного и экстремального продвижения вперед), мы не превратим его в Вавилонское Столпотворение. Вышесказанное означает, что все члены команды должны договориться об общих стандартах кодирования. Неважно каких. Правило заключается в том, что все им подчиняются. Тот, кто не желает их соблюдать, покидает команду.

13. Сорокачасовая рабочая неделя

3.8 Методология RUP

Rational Unified Process (RUP) — методология разработки программного обеспечения, созданная компанией Rational Software (рис. 3.7).

Основными принципами данной методологии являются:

1. Ранняя идентификация и непрерывное (до окончания проекта) устранение основных рисков.
2. Концентрация на выполнении требований заказчиков к исполняемой программе.
3. Ожидание изменений в требованиях, проектных решениях и реализации в процессе разработки.
4. Компонентная архитектура, реализуемая и тестируемая на ранних стадиях проекта.
5. Постоянное обеспечение качества на всех этапах разработки проекта (продукта).

6. Работа над проектом в сплочённой команде, ключевая роль принадлежит архитекторам.

Рис. 3.7 – Схема применения методологии проектирования RUP

Процессы и стадии RUP

RUP использует итеративную модель разработки. В конце каждой итерации (в идеале продолжающейся от 2 до 6 недель) проектная команда должна достичь запланированных на данную итерацию целей, создать или доработать проектные артефакты и получить промежуточную, но функциональную версию конечного продукта.

Полный жизненный цикл разработки ПО состоит из 4 этапов:

1. Начальная стадия (Inception).

На начальной стадии происходит следующее:

a) формируются видение и границы проекта;

b) создается экономическое обоснование;

c) определяются основные требования, ограничения и ключевая функциональность продукта;

d) создается базовая версия модели прецедентов;

e) оцениваются риски.

При завершении начальной фазы оценивается достижение этапа жизненного цикла цели, которое предполагает соглашение заинтересованных сторон о продолжении проекта.

2. Уточнение (Elaboration).

На данном этапе производится анализ предметной области и построение исполняемой архитектуры, а именно:

a) документирование требований;

b) проектирование, реализация и тестирование исполнимой архитектуры;

c) обновление экономических обоснований, сроков и стоимости;

d) снижение основных рисков.

Если разработка прошла успешно, значит достигнут этап жизненного цикла архитектуры.

3. Построение (Construction).

В фазе «Построение» происходит реализация большей части функциональности продукта. Данный этап завершается при первом внешнем релизе системы.

4. Внедрение (Transition).

На этапе «Внедрение» создается финальная версия продукта и передается от разработчика к заказчику, т.е. происходит бета-тестирование, обучение пользователей и определение качества продукта.

Однако, если качество продукта не соответствует ожиданиям пользователей или критериям, установленным на начальном этапе, то фаза внедрение повторяется снова. Выполнение всех целей означает достижение вехи готового продукта и завершение полного цикла разработки.

Положительные аспекты RUP:

- Учитывание изменяющихся требования, так как учесть в начале проекта все невозможно.

- Интеграция функций происходит постепенно, то есть каждая «деталь» проходит цикл разработки, проверки и внедрения в проект, благодаря этому снижаются риски и стоимость производства.
- Ранний выпуск продукта. ПО выходит с уменьшенной функциональностью, чтобы занять нишу на рынке и противостоять конкурентам, после чего обрастает «мясом».
- Повторное использование. При наращивании функциональности проще выделить типовые решения, которые сократят разработку.
- Постоянное обучение. Из-за частых итераций разработчики не имеют больших пауз между доработкой кода, поэтому профессиональный рост происходит плавно и безболезненно.
- Постоянное улучшение продукта. Итерации позволяют оценить проект не только с точки зрения соответствия плану и техническому заданию, но и найти пути увеличения эффективности и качества продукта.

3.9 Методология RAD

В 90-е годы XX века на основе спиральной модели была основана практическая технология, получившая название «быстрая разработка приложения» — RAD (Rapid Application Development) (рис. 3.8). При этом ЖЦ состоял из четырех стадий:

- анализ и планирование требований;
- проектирование;
- реализация;
- внедрение.

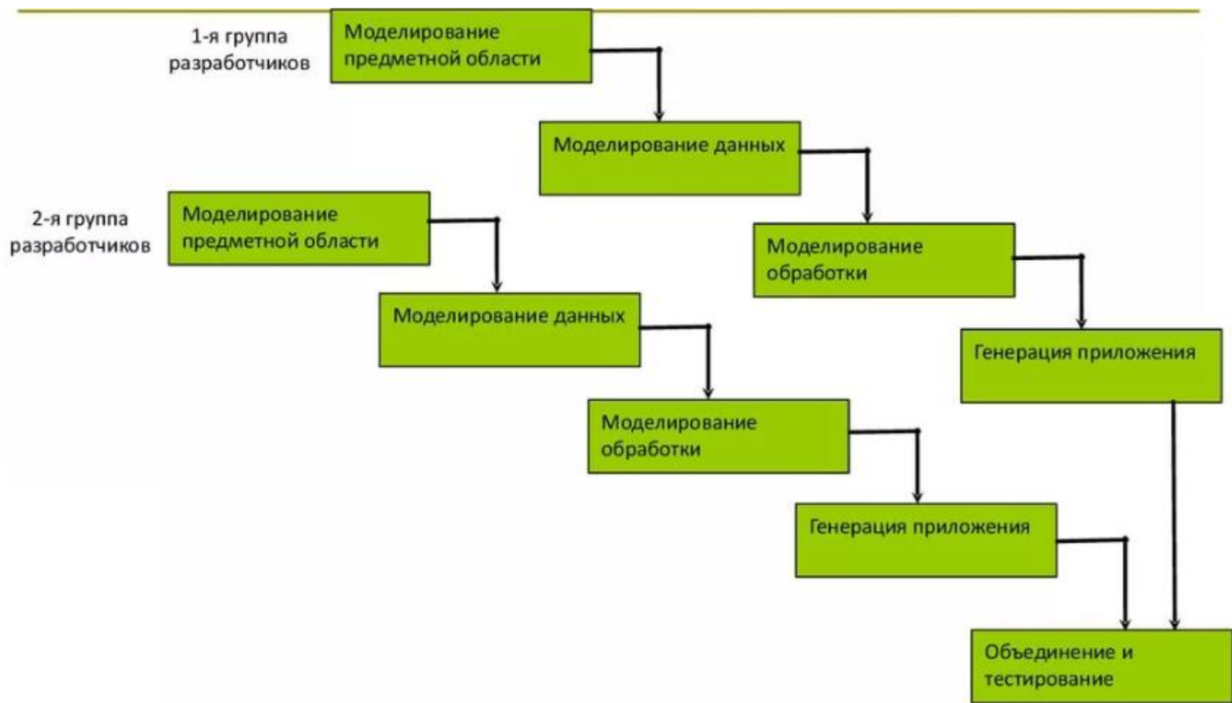

Рис. 3.8 – Схема работы RAD

Основные принципы RAD:

- разработка приложений итерациями;
- необязательность полного завершения работ на каждой из стадий жизненного цикла ПО;
- обязательность вовлечения пользователей в процесс разработки;
- применение средств управления конфигурацией, облегчающих внесение изменений в проект и сопровождение готовой системы;
- использование прототипирования, позволяющее полнее выяснить и удовлетворить потребности пользователей;
- тестирование и развитие проекта, осуществляемые одновременно с разработкой;
- ведение разработки немногочисленной хорошо управляемой командой профессионалов;
- грамотное руководство разработкой системы, четкое планирование и контроль выполнения работ.

RAD — методология, которая во главу угла ставит скорость и удобство разработки. Одно из главных условий — использование языка быстрой разработки. Это название абстрактного языка программирования, с помощью которого программист способен решать задачи быстрее, чем с представителями третьего поколения (C / C ++, Pascal или Fortran). Вот ещё несколько пунктов концепции:

- Использование фокус-групп для сбора требований.
- Прототипирование и пользовательское тестирование конструкций.
- Повторное использование программных компонентов.
- Использование плана, не включающего переработку, или дизайн следующей версии продукта.
- Проведение неформальных совещаний по запросу одной из сторон.

RAD предполагает использование целого комплекса инструментов помимо языка быстрой разработки: системы сбора требований, среды разработки, фреймворки, программы для группового общения, ПО для тестирования.

Преимущества:

1. Rad-концепция уменьшает время разработки, а задействование ранее использованных компонентов позволяет существенно ускорить рабочий процесс.
2. Все функции подразделяются на модули, с которыми работать гораздо легче.
3. Большим проектам требуются высококвалифицированные инженеры.

Недостатки:

1. И конечный пользователь, и разработчик должны быть привержены желанию завершить работу над продуктом в сокращённые сроки. Если такого желания нет, проект провалится.

2. RAD базируется на объектно-ориентированном подходе, и если возникают трудности с делением на модули, RAD может работать не очень хорошо.

3.10 Некоторые другие методологии разработки ПО

Crystal Clear

Гибкая методология Crystal Clear (иначе кристально чистое выполнение) была создана в 2004 году Алистером Коуберном. Она предназначается для небольших коллективов от 6 до 10 человек для разработки некритичных бизнес-приложений. Как и все гибкие методологии Crystal Clear опирается больше на людей, чем на процессы (рис. 3.9).

Основная идея данной методологии — каждая команда - набором людей с разным уровнем знаний, разными умениями и опытом. Из-за этого не существует универсального подхода для разработки софта, он должен определяться в процессе общения внутри группы. Там же назначаются роли, инструменты, стандарты. Затем группа принимается за единицу и те же самые вопросы решаются на уровень выше, пока иерархия не дойдет до заказчика.

В данной методологии используется 7 практик, 3 из которых обязательны.

1. Частая поставка продукта.
2. Улучшения через рефлексию.
3. Личные коммуникации.
4. Чувство безопасности.
5. Фокусировка.
6. Простой доступ к экспертам.
7. Качественное техническое окружение.

Методология Crystal Clear использует преимущества небольшого размера и расстояния между группами для усиления хорошей коммуникации до более эффективной осмотической.

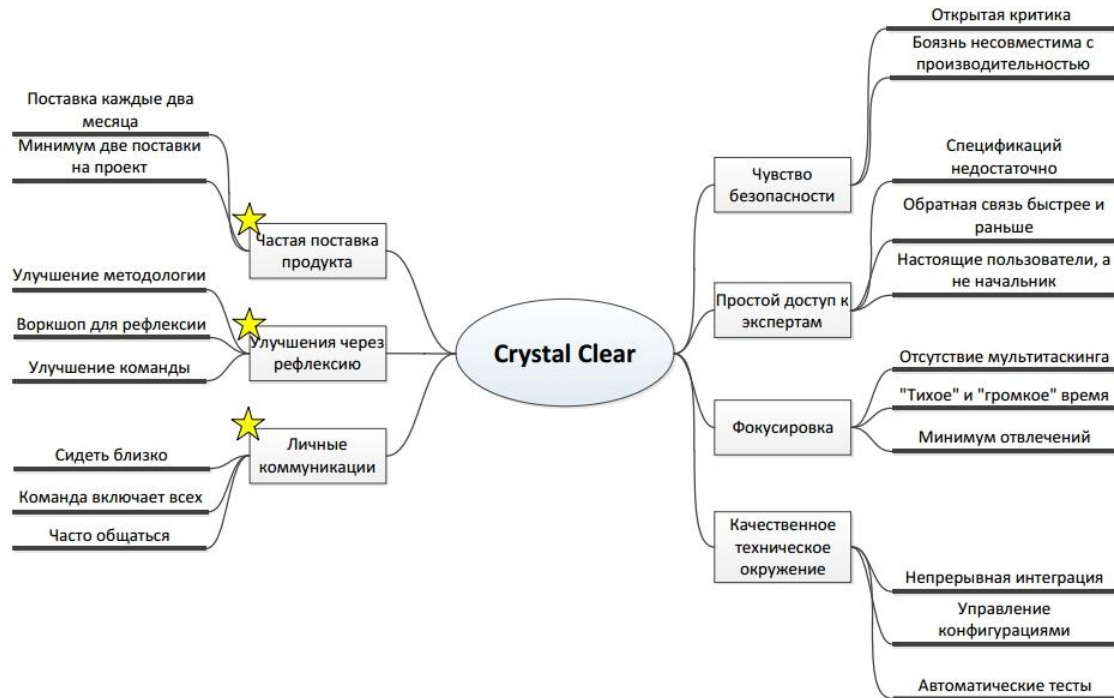

Рис. 3.9 – Разработка с использованием методологии Crystal Clear

DSDM

Модель развития динамических систем была разработана в Великобритании в середине 1990-х годов и является эволюционным развитием быстрой разработки приложений (RAD). Основная идея стандартная: при планировании в самом начале невозможно понимать всех тонкостей разработки, поэтому весь процесс — исследовательская работа.

В данной модели существует три стадии:

1. Предпроектная стадия. На ней происходит авторизация реализация проекта, определение финансовых параметров и команда.

2. Жизненный цикл проекта представляет собой реализацию проекта и включает в себя пять этапов.

3. Постпроектная стадия обеспечивает качественную эксплуатацию системы.

Жизненный цикл проекта включает в себя пять стадий (первые две фактически объединяются):

1. Определение реализуемости.
2. Экономическое обоснование.
3. Создание функциональной модели.
4. Проектирование и разработка.
5. Реализация.

В DSDM тоже присутствует деление на команды, в каждой из которых есть уполномоченный для принятия стратегических решений. В процессе могут участвовать все заинтересованные стороны: пользователи, разработчики, заказчики, руководители. Тестирование проводится на протяжении всего жизненного цикла (рис. 3.10).

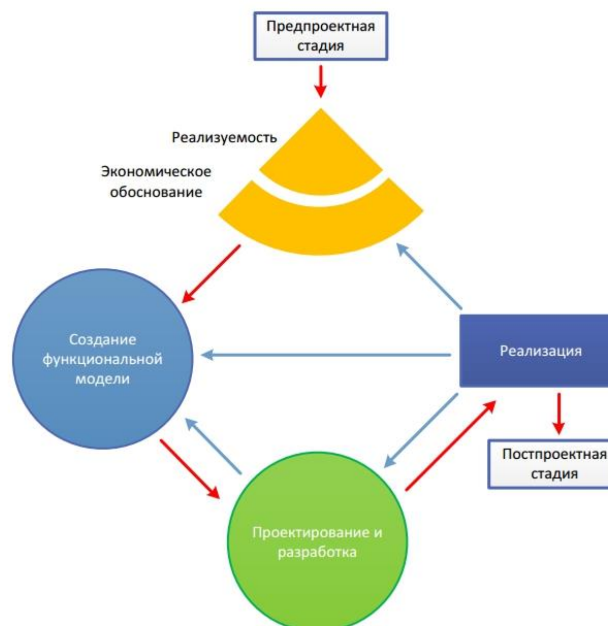

Рис. 3.10 – Общая схема DSDM

FDD

Feature driven development (FDD, разработка, управляемая функциона-льностью) — итеративная методология разработки ПО. Основная цель данной методологии — разработка реального, работающего программного обеспечения систематически, в поставленные сроки (рис. 3.11).

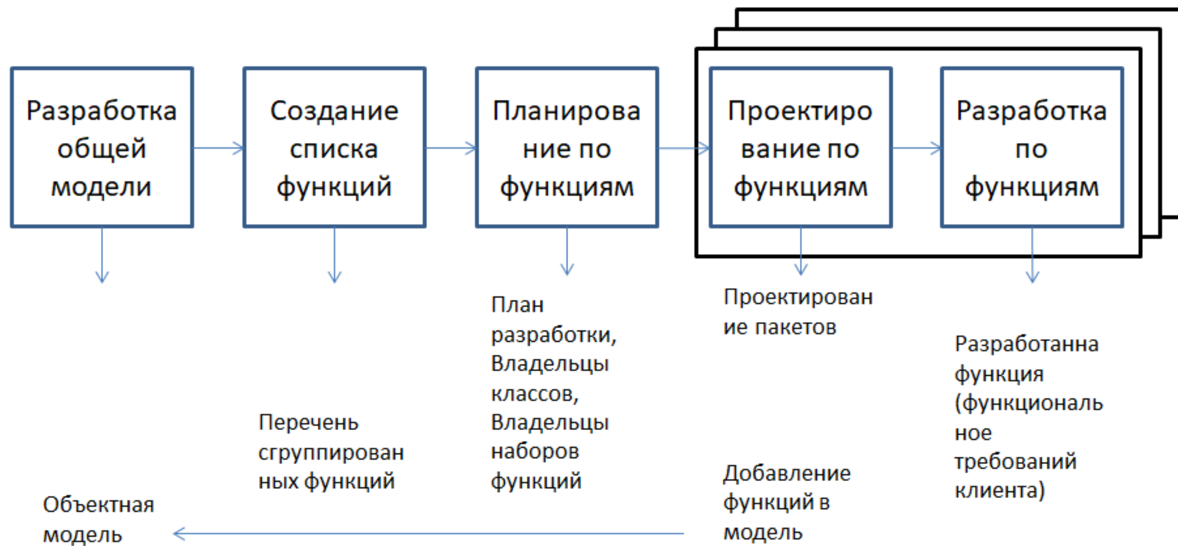

Рис. 3.11 – Схема работы FDD

FDD — процесс для обеспечения масштабируемости и повторяемости, при этом поощряющий творчество и инновации.

FDD включает в себя пять базовых видов деятельности:

1. разработка общей модели;
2. составление списка необходимых функций системы;
3. планирование работы над каждой функцией;
4. проектирование функции;
5. реализация функции.

Первые два процесса относятся к началу проекта. Последние три осуществляются для каждой функции. Разработчики в FDD делятся на «хозяев классов» и «главных программистов». Главные программисты привлекают хозяев задействованных классов к работе над очередным свойством. Работа над проектом предполагает частые

сборки и делится на итерации, каждая из которых предполагает реализацию определенного набора функций.

Также у FDD есть основные принципы:

- Разработка каждого крупного проекта должна иметь системность.
- Процессы должны быть простыми и проработанными.
- Ценность и логичность процесса должна быть ясна каждому члену команды.
- Предпочтение отдаётся коротким итеративным циклам разработки. Это уменьшает количество ошибок и позволяет быстрее наращивать функциональность.

FDD регламентирует время, которое должно затрачиваться на каждый из процессов. Организационной деятельности в цикле должна занимать не более 23−25%, в то время как на непосредственную разработку, сборку и тестирование функций необходимо тратить 75−77% времени.

JAD

JAD — это методология, нацеленная на максимальную занятость в разработке конечного пользователя. Происходит это посредством встреч и проведения совместных семинаров. JAD была придумана в 1970-х годах сотрудниками IBM и нацелена на бизнес в целом. Однако со временем данная концепция стала успешно применяться и для разработки программного обеспечения.

В отличие от подхода Waterfall, JAD приводит к сокращению времени разработки, большей удовлетворенности клиентов и экономии средств на изучении рынка. С другой стороны, это требует большой клиентской выборки и необходимости разработчиков работать не со строгими требованиями ТЗ, а с постоянно меняющимся мнением.

LD

Бережливая разработка ПО впервые была освещена в книге Мэри Поппендик и Тома Поппендика. — ещё одно ответвление гибкой методологии, предполагающее сохранение высокого морально-функционального состояния разработчиков. Это выражается в:

- Поощрении сотрудников за успешную работу.
- Изменении текущих задач только по мере необходимости или по запросу заказчика.
- Строгом выполнении плана: всё, что сверх — считается потерями времени и ресурсов.
- Внедрении общей концепции «Мыслить широко, делать мало, ошибаться быстро, учиться стремительно».

Принципы:

- **Исключение потерь**. Потерями считается всё, что не добавляет ценности для потребителя. В частности, излишняя функциональность; ожидание (паузы) в процессе разработки; нечёткие требования; бюрократизация; медленное внутреннее сообщение.
- **Акцент на обучении.** Короткие циклы разработки, раннее тестирование, частая обратная связь с заказчиком.
- **Предельно отсроченное принятие решений.** Решение следует принимать не на основе предположений и прогнозов, а после открытия существенных фактов.
- **Предельно быстрая доставка заказчику.** Короткие итерации.
- **Мотивация команды.** Нельзя рассматривать людей исключительно как ресурс. Людям нужно нечто большее, чем просто список заданий.
- **Интегрирование.** Передать целостную информацию заказчику. Стремиться к целостной архитектуре. .

- **Целостное видение.** Стандартизация, установление отношений между разработчиками. Разделение разработчиками принципов бережливости. «Мыслить широко, делать быстро, ошибаться мало; учиться стремительно».

Вопросы к главе 3

1. Что такое методологии разработки программного обеспечения?
2. Приведите классификацию методологий разработки ПО?
3. Какие классические методологии разработки ПО Вы знаете?
4. Какие гибкие методологии разработки ПО Вы знаете?
5. Перечислите основные идеи каскадной методологии разработки ПО.
6. Перечислите основные этапы каскадной методологии разработки ПО.
7. Перечислите преимущества и недостатки каскадной методологии разработки ПО.
8. В чем заключаются основные особенности спиральной модели разработки ПО?
9. В чем заключаются основные особенности итерационной модели разработки ПО?
10. В каких случаях рекомендуется использовать гибкие методологии разработки ПО?
11. Перечислите основные принципы Agile.
12. Как основные принципы Agile реализуются в Scrum?
13. Перечислите основные принципы экстремального программирования.
14. Опишите методологию RUP.
15. Как методология RAD связана с другими методологиями разработки ПО?

Глава 4. ТЕСТИРОВАНИЕ КАК ЧАСТЬ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

4.1 Роль тестирования ПО

Системы с ПО - неотъемлемая часть нашей жизни, от бизнес-приложений (таких, как банковское ПО) до потребительских товаров (таких, как автомобили). Многие люди имели опыт использования программного обеспечения, которое не работало так, как от него ожидалось. Программное обеспечение, которое не работает корректно, может привести ко многим проблемам, включая потерю денег, времени или деловой репутации, и стать причиной травмы или смерти.

Человек может сделать ошибку (просчет), которая порождает дефект (недочет, помеху) в программном коде или документе. Если код с дефектом выполнен, то система может быть не в состоянии сделать то, что должна делать (или сделать то, что от нее не ожидают), порождая отказ. Дефекты в программном обеспечении, системах или документах могут в результате привести к отказам, но не все дефекты дают такой результат.

Дефекты встречаются, потому что люди склонны ошибаться, существует нехватка времени, сложность кода, сложность инфраструктуры, изменения технологий и /или много системных взаимодействий.

Отказы так же могут быть вызваны условиями окружающей среды. Например, радиация, электромагнитные поля и загрязнения могут вызвать отказ в программно-аппаратных средствах или повлиять на выполнение программного обеспечения, изменяя условия работы аппаратных средств.

Доскональное тестирование систем и документации может уменьшить риск возникновения проблем во время функционирования

и способствует повышению качества системы программного обеспечения, если найденные дефекты исправлены прежде, чем система передана в эксплуатацию.

Тестирование программного обеспечения также может быть требованием для удовлетворения контракту или требованиям законодательства, или специализированным промышленным стандартам.

Тестирование - это возможный способ оценки качества программного обеспечения в терминах найденных дефектов, как для функциональных требований, так и для нефункциональных требований и характеристик программного обеспечения (например, надежность, практичность, эффективность, сопровождаемость и переносимость).

Тестирование может породить уверенность в качестве программного обеспечения, если не найдены или найдено немного дефектов. Должным образом разработанный тест, который пройден успешно, уменьшает общий уровень риска в системе. Когда во время тестирования находятся ошибки, качество систем программного обеспечения повышается, если эти дефекты исправлены.

Следует извлекать уроки из предыдущих проектов. Понимая первопричины дефектов, найденных в других проектах, можно улучшить процессы, что в свою очередь должно предотвратить повторное проявление этих дефектов и, как следствие, повышение качества будущих систем. Это и есть подход к обеспечению качества.

Тестирование также является деятельностью по обеспечению качества (наравне с разработкой стандартов, обучением и анализом дефектов).

Для принятия решения о достаточном объеме тестирования, необходимо принимать во внимание уровень рисков, включая технические риски, риски безопасности и бизнес риски, а также проектные ограничения, такие как время и бюджет.

Тестирование должно предоставить достаточную информацию заинтересованным лицам, чтобы принять обоснованные решения о передаче программного обеспечения или системы, прошедшей тестирование, на следующий шаг разработки или передачи клиентам.

Бытует мнение, что тестирование состоит только из прогона тестов, то есть выполнения самой программы. Но это только часть тестирования, и далеко не все, что в него входит.

Активности в тестировании существуют как до, так и после выполнения самих тестов. В эти активности входят планирование и управление, выбор тестовых условий, разработка и выполнение тестовых сценариев, проверка результатов, оценка критериев выхода, создание отчетов о процессе тестирования и об испытываемой системе и закрытие или завершающие действия после того, как фаза тестирования была выполнена. Тестирование также включает рецензирование документации (включая исходный код) и проведение статического анализа.

И динамическое, и статическое тестирования используются для достижения аналогичных целей, предоставляя информацию, которая может способствовать улучшению, как испытываемой системы, так и процессов разработки и тестирования.

Цели тестирования:

- обнаружение дефектов;
- повышение уверенности в уровне качества;
- предоставление информации для принятия решений;
- предотвращение дефектов.

Цели процессов и действия, связанные с проектированием тестов на раннем этапе жизненного цикла программного обеспечения (например, при компонентном, интеграционном и системном тестировании), могут помочь предотвратить попадание дефектов в код. Рецензирование документов (например, требований),

идентификация и разрешение проблем также помогают предотвратить появление дефектов в коде.

Разные точки зрения в тестировании преследуют разные цели. Например, в тестировании на этапе разработки (таком, как компонентное, интеграционное и системное тестирование), основная цель может заключаться в том, чтобы вызвать как можно больше отказов, чтобы дефекты в программном обеспечении были идентифицированы и могли быть исправлены. В приемочном тестировании основная цель может состоять в том, чтобы подтвердить, что система работает, как ожидалось и повысить уверенность в том, что она удовлетворяет требованиям. В некоторых случаях основная цель тестирования может состоять в том, чтобы оценить качество программного обеспечения (без намерения исправлять дефекты) и дать информацию заинтересованным лицам о рисках выпуска системы в установленный срок. Тестирование в период сопровождения в основном заключается в проверке отсутствия новых дефектов, которые могли попасть во время разработки изменений. Во время эксплуатационного тестирования основная цель может заключаться в том, чтобы оценить системные характеристики, такие как надежность или доступность.

Стоит различать отладку и тестирование. Динамическое тестирование может выявить отказы, вызванные дефектами. Отладка – это действия разработчиков, которые находят, анализируют и устраняют причину отказа. Повторное тестирование гарантирует, что изменение действительно предотвращает отказ. Ответственность за тестирование обычно несут тестировщики, а за отладку – разработчики.

4.2 Принципы тестирования

Эти принципы тестирования были предложены в последние 40 лет и являются общим руководством для тестирования в целом.

Принцип 1 – Тестирование демонстрирует наличие дефектов.

Тестирование может показать, что дефекты присутствуют, но не может доказать, что их нет. Тестирование снижает вероятность наличия дефектов, находящихся в программном обеспечении, но, даже если дефекты не были обнаружены, это не доказывает его корректности.

Принцип 2 – Исчерпывающее тестирование недостижимо.

Полное тестирование с использованием всех комбинаций вводов и предусловий физически невыполнимо, за исключением тривиальных случаев. Вместо исчерпывающего тестирования должны использоваться анализ рисков и расстановка приоритетов, чтобы более точно сфокусировать усилия по тестированию.

Принцип 3 – Раннее тестирование.

Чтобы найти дефекты как можно раньше, активности по тестированию должны быть начаты как можно раньше в жизненном цикле разработки программного обеспечения или системы, и должны быть сфокусированы на определенных целях.

Принцип 4 – Скопление дефектов.

Усилия тестирования должны быть сосредоточены пропорционально ожидаемой, а позже реальной плотности дефектов по модулям. Как правило, большая часть дефектов, обнаруженных при тестировании или повлекших за собой основное количество сбоев системы, содержится в небольшом количестве модулей.

Принцип 5 – Парадокс пестицида.

Если одни и те же тесты будут прогоняться много раз, в конечном счете этот набор тестовых сценариев больше не будет находить новых дефектов. Чтобы преодолеть этот “парадокс

пестицида”, тестовые сценарии должны регулярно рецензироваться и корректироваться, новые тесты должны быть разносторонними, чтобы охватить все компоненты программного обеспечения, или системы, и найти как можно больше дефектов.

Принцип 6 – Тестирование зависит от контекста.

Тестирование выполняется по-разному в зависимости от контекста. Например, программное обеспечение, в котором критически важна безопасность, тестируется иначе, чем сайт электронной коммерции.

Принцип 7 – Заблуждение об отсутствии ошибок.

Обнаружение и исправление дефектов не помогут, если созданная система не подходит пользователю и не удовлетворяет его ожиданиям и потребностям.

4.3 Основной процесс тестирования

Активность тестирования, которую легче всего увидеть – это выполнение тестов. Но чтобы быть эффективным и рациональным, планы тестирования должны включать время, которое будет потрачено на планирование тестов, разработку тестовых сценариев, подготовку к выполнению и оценку результатов.

Основной процесс тестирования состоит из следующих направлений деятельности:

- планирование и управление;
- анализ и проектирование;
- внедрение и реализация;
- оценка критериев выхода и создание отчетов;
- действия по завершению тестов.

Несмотря на логическую последовательность, действия в процессе могут накладываться друг на друга или происходить

одновременно. Для конкретных системы и проекта обычно требуется адаптация этих направлений деятельности.

Планирование и управление тестированием

Планирование тестирования – это действия, направленные на определение целей тестирования и описание задач тестирования для достижения этих целей и миссии.

Управление тестированием – это постоянное сопоставление текущего положения дел с планом и отчетность о состоянии дел, включая отклонения от плана. Это позволяет предпринять меры, необходимые для достижения миссии и целей проекта. Чтобы обеспечить управление тестированием, активности тестирования должны проверяться в течение всего проекта. Планирование тестирования учитывает данные, полученные при проверке и управлении.

Анализ и проектирование тестов

Анализ и проектирование тестов - это деятельность, во время которой общие цели тестирования материализуются в тестовые условия и тестовые сценарии.

Для анализа и проектирования тестов поставлены следующие основные задачи:

- Рецензирование базиса тестирования (такого, как требования, уровень целостности программного обеспечения 1 (уровень риска), отчеты об анализе рисков, архитектура, дизайн, технические требования к интерфейсу).
- Оценка тестируемости базиса тестирования и объектов тестирования.
- Идентификация и расстановка приоритетов условий тестирования, основанных на анализе элементов тестирования, спецификации, поведении и структуры программного обеспечения.

• Разработка и расстановка приоритетов тестовых сценариев высокого уровня.

• Выявление необходимых данных для поддержки тестовых условий и тестовых сценариев.

• Проектирование и установка тестового окружения и выявление необходимой инфраструктуры и инструментов.

• Создание двунаправленной трассируемости между тестовым базисом и тестовым сценарием.

Реализация и выполнение тестов

Реализация и выполнение тестов – это деятельность, где процедуры тестирования или автоматизированные сценарии задаются последовательностью тестовых сценариев, а также собирается любая информация, необходимая для выполнения тестов, разворачивается окружающая среда, и запускаются тесты.

Для реализации и выполнения тестов поставлены следующие основные задачи:

• Завершение, реализация и расстановка приоритетов тестовых сценариев (включая проектирование тестовых данных).

• Разработка и расстановка приоритетов процедур тестирования, создание тестовых данных и, если потребуется, подготовка тестовых обвязок и написание автоматизированных сценариев тестирования.

• Создание тестовых наборов на основе процедур тестирования для эффективного выполнения тестов.

• Проверка правильности настройки тестового окружения.

• Проверка и обновление двунаправленной трассируемости между тестовым базисом и тестовым сценарием.

• Выполнение процедур тестирования либо вручную, либо используя инструменты выполнения тестов, согласно заданному плану.

• Регистрация результатов выполнения тестов и запись наименований и версий объекта тестирования, тестовых инструментов и тестового обеспечения.

• Сравнение фактических и ожидаемых результатов.

• Отчет о несоответствиях как об инцидентах и их анализ для установки причины (например, дефект в коде, в конкретных тестовых данных, в тестовом документе, или ошибка выполнения теста).

• Повторение тестовых действий, результаты которых привели к каждому из несоответствий. Например, повторное выполнение теста, который ранее не прошел, чтобы подтвердить исправление (подтверждающее тестирование), выполнение исправленных тестов и/или повторное выполнения тестов с целью убедиться, что дефекты не появились в той области программного обеспечения, которая не изменялась, или что исправление дефекта не повлекло за собой других дефектов (регрессионное тестирование).

Оценка критериев выхода и отчетность

Оценка критериев выхода - это деятельность, где выполнение тестов оценивается согласно определенным целям. Она должна быть выполнена для каждого уровня тестирования (см. раздел 2.2).

Для оценки критериев выхода поставлены следующие основные задачи:

• сверка протокола тестирования в сравнении с критериями выхода, определенными в плане тестирования;

• анализ необходимости использования дополнительных тестов или изменения критериев выхода;

• написание итогового отчета о тестировании для заинтересованных лиц.

Действия по завершению тестирования

Действия по завершению тестирования собирают данные о завершенных испытаниях для объединения опыта, тестового обеспечения, фактов и цифр. Действия по завершению тестирования происходят на тех этапах проекта, когда система программного обеспечения выпущена, тестирование завершено (или прервано), этап был завершен, или релиз по сопровождению был закончен.

Для действий по завершению тестирования поставлены следующие основные задачи:

- Проверка, что запланированные результаты достигнуты.
- Закрытие отчетов об инцидентах или внесение изменений в записи по каждому из открытых инцидентов.
- Документирование приемки системы.
- Завершение и архивирование тестового обеспечения, тестового окружения и инфраструктуры тестирования для последующего использования.
- Передача тестового обеспечения организации сопровождения.
- Анализ полученных уроков для определения изменений, необходимых для будущих релизов и проектов.
- Использование собранной информации для повышения зрелости процесса тестирования.

4.4 Уровни тестирования ПО

Для каждого уровня тестирования может быть определено: цели, артефакты процесса разработки, на основании которых будут разработаны тестовые сценарии, объекты тестирования, типичные дефекты и отказы, которые могут быть найдены во время тестирования, требования к тестовой обвязке, инструментарий и ответственность участников.

Конфигурационные данные для тестирования системы нужно рассматривать во время планирования тестирования, если такие данные необходимы.

Компонентное тестирование

Базис тестирования:

• требования к компонентам;

• детальный дизайн;

• код.

Типичные объекты тестирования:

• компоненты;

• программы;

• программы конвертации и миграции данных;

• модули БД.

Компонентное тестирование (также известное как модульное) занимается поиском дефектов и верификацией функционирования программных модулей, программ, объектов, классов и т.п., которые можно протестировать изолированно. Это может быть сделано изолированно от остальной части системы, в зависимости от контекста ЖЦ разработки и системы. В процессе могут быть использованы заглушки, драйвера и эмуляторы.

Компонентное тестирование может включать как тестирование функциональности и специфичных нефункциональных характеристик, таких как поведение ресурсов (например, поиск утечки памяти) или тестирование надежности, так и структурное тестирование (например, покрытие кода). Тестовые сценарии разрабатываются на основе артефактов процесса разработки, таких как спецификация компонентов, дизайн или модель БД.

Обычно, компонентное тестирование производится с доступом к тестируемому коду и с поддержкой рабочего окружения, такого как фреймворк модульного тестирования или утилиты отладки. На практике компонентное тестирование обычно производится

разработчиками, которые пишут код. Дефекты обычно исправляются сразу после того, как становятся известны, без занесения их в базу дефектов.

Один из подходов к компонентному тестированию – составить автоматизированные тестовые сценарии до кодирования. Это называется разработкой, управляемой тестированием. Этот подход состоит из множества итераций и основывается на циклах разработки тестовых сценариев, написании и интеграции небольшого участка кода и выполнении компонентного тестирования, корректируя любые проблемы и выполняя тесты, пока не будет получен положительный результат.

Интеграционное тестирование

Базис тестирования:

• проект системы;

• архитектура;

• бизнес-процессы;

• сценарии использования.

Типичные объекты тестирования:

• БД подсистем;

• инфраструктура;

• интерфейсы.

Конфигурация системы:

• Конфигурационные данные.

Интеграционное тестирование проверяет интерфейсы между компонентами, взаимодействие различных частей системы, таких как операционная системы, файловая система, аппаратное обеспечение, и интерфейсы между системами.

Интеграционное тестирование может состоять из одного или более уровней и может быть выполнено на тестовых объектах разного размера следующим образом:

1. Компонентное интеграционное тестирование проверяет взаимодействие между программными компонентами и производится после компонентного тестирования.

2. Системное интеграционное тестирование проверяет взаимодействие между системами или между аппаратным обеспечением и может быть выполнено после системного тестирования. В этом случае, разработчики могут управлять только одной стороной интерфейса. Однако, это может рассматриваться как риск. Бизнес-процессы могут включать последовательность систем; могут быть важны кроссплатформенные различия.

Чем больше объем интеграции, тем труднее становится изолировать дефекты отдельного компонента или системы, которые могут привести к увеличению рисков и требующих дополнительного времени для решения проблем.

Стратегии системного интеграционного тестирования могут основываться на архитектуре системы (такой как нисходящая или восходящая), функциональных задачах, последовательности обработки транзакций или других аспектах системы и ее компонентов. Для того, чтобы упростить процесс изоляции отказа и как можно раньше обнаруживать дефекты, интеграция должна проводиться по возрастающей, а не происходить по сценарию «большого взрыва».

Тестирование специфичных нефункциональных характеристик (например, производительности), может быть включено в интеграционное тестирование, наравне с функциональным.

На каждой стадии интеграции тестировщики концентрируют все внимание именно на интеграции как таковой. Например, если интегрируется модуль А с модулем В, они проверяют взаимодействие модулей, а не функциональность каждого из них, т.к. она должна быть проверена во время компонентного тестирования. Для

тестирования могут использоваться как функциональный, так и структурный подходы.

В идеале, тестировщики должны понимать архитектуру и ее влияние на интеграционное планирование. Если интеграционное тестирование планируется до разработки компонентов или системы, эти компоненты могут разрабатываться в порядке, обеспечивающем наиболее эффективное тестирование.

Системное тестирование

Базис тестирования:

• система и спецификация требований к программному обеспечению;

• сценарии использования;

• функциональная спецификация;

• отчеты об анализе степени риска.

Типичные объекты тестирования:

• руководство по эксплуатации системы;

• конфигурация системы.

Конфигурационные данные

Системное тестирование сконцентрировано на поведении тестового объекта как целостной системы или продукта. Область тестирования должна быть четко определена в главном плане тестирования либо плане тестирования для конкретного уровня тестирования.

Во время системного тестирования тестовое окружение должно быть как можно ближе к предполагаемому эксплуатационному окружению системы для минимизации риска пропуска отказов, связанных с эксплуатационным окружением системы.

Системное тестирование может включать тесты, основанные на рисках или спецификациях требований, бизнес-процессах, сценариях использование системы, или других высокоуровневых текстовых

описаниях или моделях поведения системы, взаимодействия с ОС и системными ресурсами.

Системное тестирование должно заниматься исследованием функциональных и нефункциональных требований к системе и качеством обрабатываемых данных. Тестировщики также должны уметь выполнять свои обязанности в случае неполных или недокументированных требований. Системное тестирование функциональных требований начинается с тестирования на основе спецификаций (тестирования методом черного ящика), для различных аспектов системы. Например, таблица решений может быть создана на основе бизнес-правил работы системы. Тестирование на основе структуры (тестирование методом белого ящика) может использоваться для оценки тщательности тестирования того или иного структурного элемента, такого как структура меню или навигация веб-страницы.

Системное тестирование чаще всего выполняет независимая тестовая команда.

Приемочное тестирование

Базис тестирования:

• пользовательские требования;
• системные требования;
• сценарии использования;
• бизнес процессы;
• отчеты об анализе степени риска.

Типичные объекты тестирования:

• бизнес-процессы на полностью интегрированной системе;
• процессы эксплуатации и обслуживания;
• процедуры использования;
• форы;
• отчеты.

Конфигурационные данные

Приемочным тестированием системы чаще всего занимаются заказчики или пользователи системы, а также другие заинтересованные лица.

Основная цель приемочного тестирования – проверка работоспособности системы, частей системы или отдельных нефункциональных характеристик системы. Главной целью приемочного тестирования является поиск дефектов.

Приемочное тестирование оценивает готовность системы к развертыванию и использованию, хотя это не обязательно самый последний уровень тестирования. Например, крупномасштабные тесты по системной интеграции можно провести именно во время приемочного тестирования системы.

Приемочное тестирование может проводиться в различные моменты ЖЦ разработки, например:

• Для коробочного продукта приемочное тестирование можно провести при установке или интеграции.

• Приемочное тестирование удобства использования компонента можно провести во время компонентного тестирования.

• Приемочное тестирование новой функциональности можно проводить до системного тестирования.

Типичные виды приемочного тестирования:

Пользовательское приемочное тестирование

Обычно проверяет готовность системы для использования в бизнесе.

Эксплуатационное (приемочное) тестирование

Приемочное тестирование, проводимое системными администраторами, включает:

• тестирование резервного копирования \ восстановления;

• восстановление после сбоев;

• управление пользователями;

• задачи сопровождения;

• задачи загрузки и миграции данных;

• периодическая проверка уязвимостей системы.

Контрактное и правовое приемочное тестирование

Контрактное приемочное тестирование выполняется для проверки требований, предъявляемых контрактом в к разрабатываемому ПО. Критерий приема должен быть определен непосредственно в контракте. Приемочное тестирование на соответствие стандартам выполняется для проверки соответствия стандартам государственным, юридическим или стандартам безопасности.

Альфа и бета тестирование (или тестирование в условиях эксплуатации)

Разработчики рыночного, или коробочного, ПО часто хотят получить отзывы от потенциальных или существующих заказчиков до того, как начнется продажа продукта. Альфа тестирование выполняется организацией, разрабатывающей продукт, но не группой разработчиков. Бета тестирование, или тестирование в условиях эксплуатации, выполняется покупателями или потенциальными заказчиками на их собственных мощностях.

В организации могут использоваться и другие термины приемочного тестирования, такие как производственное приемочное тестирование и стороннее приемочное тестирование для систем, которые проверяются до и после установки на стороне заказчика.

Типы тестирования

Группы активностей тестирования могут быть направлены на проверку работоспособности системы (или части системы), принимая за основу различные цели и причины для тестирования.

Типы тестирования определяются целями тестирования, которые могут быть следующими:

• функция, выполняемая программой;

• нефункциональная характеристика качества, такая как надежность или удобство использования;

• структура или архитектура программы или системы.

Подтверждение изменений, т.е. подтверждение, что дефект был исправлен (подтверждающее тестирование) и поиск непреднамеренных изменений (регрессионное тестирование).

Модель программного обеспечения может быть разработана и\или использована во время структурного (например, модель потока управления или модель структуры меню), нефункционального тестирования, например, модель нагрузки, модель удобства использования, модель угроз безопасности) и функционального тестирования (например, модель бизнес-процессов, модель переходов состояний или спецификация не естественном языке).

Тестирование функций (функциональное тестирование)

Функции, которые выполняет система, подсистема или компонент, могут быть описаны в таких артефактах процесса разработки как спецификация требований, сценарии использования системы или функциональная спецификация, либо могут быть недокументированны. Эти функции описывают, «что» эта система делает.

Функциональные тесты разрабатываются на основе функций и возможностей системы (описанных в документах или понятных тестировщикам) и их взаимодействия со специфичными системами и могут быть выполнены на всех уровнях тестирования (например, тесты для компонентов могут основываться на спецификациях компонентов).

Методы разработки тестов на основе спецификаций используются для извлечения информации о тестовых условиях и тестовых сценариях из функциональности программы или системы. Функциональное тестирование рассматривает внешнее поведение программного обеспечения (тестирование методом черного ящика).

Один из типов функционального тестирования, тестирование безопасности, исследует функции (например, брандмауэр) касающиеся обнаружения угроз, таких как вирусы, поступающих извне. Другой тип функционального тестирования, тестирование возможности взаимодействия, оценивает способность программного продукта взаимодействовать с одним или более указанными компонентами или системами.

Тестирование нефункциональных характеристик (нефункциональное тестирование)

Нефункциональное тестирование включает, но не ограничивается, нагрузочное тестирование, тестирование производительности, стресс-тестирование, тестирование удобства использования, тестирование восстановления, тестирование надежности и тестирование переносимости. Это тестирование того, «как» система работает.

Нефункциональное тестирование может выполняться на всех уровнях тестирования. Термин нефункциональное тестирование описывает тесты, необходимые для оценки характеристик систем и программ, которые могут быть количественно измерены, такие как время отклика при тестировании производительности. Эти тесты могут ссылаться на модели качества, такие как «Разработка программного обеспечения – Качество программного продукта» (ISO 9126). Нефункциональное тестирование рассматривает внешнее поведение программного обеспечения и в большинстве случаев использует разработку тестов методом черного ящика.

Тестирование структуры/архитектуры программного обеспечения (структурное тестирование)

Структурное тестирование (тестирование методом белого ящика) может выполняться на всех уровнях тестирования. Структурные методы тестирования лучше всего использовать после методов разработки тестов на основе спецификации, чтобы измерить

тщательность тестирования, используя измерения покрытия структуры программы.

Покрытие – это часть структуры программы, которая была охвачена тестированием, выраженная в процентах. Если покрытие не равно 100%, то необходимо разрабатывать дополнительные тесты для покрытия пропущенных участков программы. Методики покрытия описаны в главе 4.

На всех уровнях тестирования, особенно в компонентном и компонентном интеграционном тестировании, могут использоваться инструментальные средства для измерения покрытия кода. Структурное тестирование может основываться на архитектуре системы, такой как иерархия вызовов.

Методы структурного тестирования могут быть применены на системном, системном интеграционном и приемочном уровнях (например, применены к бизнес-моделям или структурам меню).

Тестирование изменений: подтверждающее и регрессионное тестирование

После того, как дефект обнаружен и исправлен, программу необходимо перепроверить, чтобы убедиться, что исходный дефект успешно устранен. Это называется подтверждением. Отладка (локализация и исправление дефекта) относится к процессу разработки, а не тестирования.

Регрессионное тестирование – это повторное тестирование уже протестированных программ после внесения в них изменений, чтобы обнаружить дефекты, внесенные или пропущенные в результате этих действий. Эти дефекты могут быть как в проверяемом компоненте, так и в связанном или несвязанным с ним. Регрессионное тестирование выполняется, когда в программное обеспечение или его окружение вносятся изменения. Глубина регрессионного тестирования оценивается риском пропуска дефектов в программном обеспечении, которое работало ранее.

Тесты должны быть повторяемыми, если они должны использоваться для подтверждающего или регрессионного тестирования.

Регрессионное тестирование может выполняться на всех уровнях тестирования и включает функциональное, нефункциональное и структурное тестирование. Регрессионные наборы тестов запускаются множество раз и меняются медленно, поэтому регрессионное тестирование является хорошим кандидатом на автоматизацию.

4.5 Организация и независимость тестирования

Эффективность поиска дефектов и рецензирования может быть повышена с помощью независимых тестировщиков, Варианты независимости могут быть следующими:

- отсутствие независимых тестировщиков: разработчики тестируют собственный код;
- независимые тестировщики в команде разработчиков;
- независимая команда или группа тестирования в организации, отчитывающаяся менеджеру проекта или исполнительному менеджеру;
- независимые тестировщики из бизнес-организации или сообщества пользователей;
- независимые специалисты тестирования для отдельных типов тестирования, например, тестировщики удобства использования, тестировщики безопасности или тестировщики сертификации (которые сертифицируют ПО на соответствие стандартам и правилам);
- независимые тестировщики, привлеченные на аутсорсинг или сторонние по отношению к организации.

• для больших, сложных или критичных с точки зрения безопасности проектов обычно лучше иметь несколько уровней тестирования, при этом некоторые или все уровни выполняются независимыми тестировщиками. Разработчики также могут участвовать в тестировании, особенно на низких уровнях, но недостаток объективности зачастую ограничивает их эффективность. Независимые тестировщики могут иметь право определять правила и процессы тестирования, но принимать роли в процессах должны только после недвусмысленного разрешения на это.

Преимущества независимого тестирования:

• независимые тестировщики беспристрастны, видят другие, отличные дефекты;

• независимые тестировщики могут проверять предположения, сделанные во время создания спецификаций и разработки системы.

Недостатки независимого тестирования:

• изолированность от команды разработчиков (в случае полной независимости тестировщиков);

• разработчики теряют чувство ответственности за качество;

• независимые тестировщики могут быть узким местом, их могут обвинить в задержке выпуска продукта.

Задачи тестирования могут выполняться людьми в специальной тестовой роли или кем-то в другой роли, например, менеджером проекта, менеджером по качеству, разработчиком, экспертом в бизнесе или предметной области, инфраструктуре или ИТ.

4.6 Мониторинг тестирования и контроль тестирования

Мониторинг прогресса тестирования

Целью мониторинга тестирования является предоставление результата и обзора процесса тестирования. Информация отслеживается вручную или автоматически и может быть

использована для измерения критериев выхода, таких как покрытие. Метрики также могут быть использованы для оценки прогресса тестирования по сравнению с запланированным расписанием и бюджетом.

Обычные тестовые метрики включают в себя:

• процент проделанной работы по подготовке тестовых сценариев (или процентное соотношение запланированных и подготовленных сценариев);

• процент проделанной работы по подготовке тестового окружения;

• выполнение тестовых сценариев (например, количество выполненных\невыполненных тестовых сценариев, успешно пройденных\неудачных тестовых сценариев);

• информация о дефектах (например, плотность дефектов, количество найденных и исправленных дефектов, интенсивность отказов и результаты повторного тестирования);

• тестовое покрытие требований, рисков или кода;

• субъективная уверенность тестировщиков в продукте;

• даты контрольных точек тестирования;

• стоимость тестирования, включая стоимость по сравнению с выгодой нахождения следующего дефекта или запуска следующего теста.

Отчетность по тестированию

Отчеты о тестировании предоставляют итоговую информацию о тестировании, включая:

• что произошло во время цикла тестирования, например, даты, когда были достигнуты критерии выхода;

• проанализированную информацию и метрики для поддержки рекомендаций и решений о последующих действиях, таких как оценка оставшихся дефектов, экономическое обоснование продолжения

тестирования, оставшиеся риски и уровень уверенности в тестируемом ПО.

Структура отчета о результатах тестирования приводится в «Стандарте по тестовой Документации для Программного Обеспечения» (IEEE Std 829-1998).

Метрики, которые необходимо собрать во время и в конце уровня тестирования:

- соответствие целей тестирования уровню тестирования;
- адекватность выбора подхода к тестированию;
- эффективность тестирования в отношении установленных целей.

Контроль тестирования

Контроль тестирования описывает любые направляющие или корректирующие действия, принятые как результат по полученной и собранной информации и значениям метрик. Контроль тестирования может затрагивать любые действия по тестированию, а также воздействовать на другие действия и задачи жизненного цикла ПО.

Примеры действий по контролю тестирования:

- принятие решений на основании данных мониторинга тестирования;
- повторная расстановка приоритетов при возникновении установленного риска (например, задержка выпуска ПО);
- изменение графика тестирования согласно доступности тестового окружения;
- установка критерия входа, требующего повторного (подтверждающего) тестирования исправлений, сделанных разработчиком, перед принятием их в сборку.

Вопросы к главе 4

1. Зачем требуется проводить тестирование программного обеспечения?
2. Дайте определение понятия тестирование ПО.
3. Каковы цели процесса тестирования ПО?
4. В чем разница между процессами отладки и тестирования?
5. Перечислите основные принципы тестирования.
6. Из каких направлений деятельности состоит процесс тестирования?
7. Как осуществляется анализ и проектирование тестов?
8. Перечислите уровни тестирования ПО.
9. Что такое компонентное тестирование?
10. Что такое интеграционное тестирование?
11. Что такое системное тестирование?
12. Что такое приемочное тестирование?
13. В чем разница между альфа- и бета- тестированием?
14. Что такое регрессионное тестирование?
15. Как обеспечивается независимость тестирования?
16. Как осуществляется мониторинг и контроль тестирования?

Глава 5. ОЦЕНКА В РАЗВИТИИ ПРОГРАММНОГО ПРОЕКТА

5.1 Роль оценки в планировании проекта

Разработка крупных информационных систем (ИС) процесс трудоёмкий, требующий комплексного решения проблем времени, бюджета, и самой функциональности разрабатываемой системы.

Зачастую проекты завершаются не в срок, бюджет проекта превышает первоначально заданный и т.д. - примеров можно привести множество. Как представляется, причинами подобных негативных результатов являются:

- Недостаточно продуманный план менеджера проекта, иными словами неправильное управление проектом.
- Неполная или неточная спецификация на проект, а также требования заказчика, которые меняются в процессе разработки.
- Некачественная оценка проекта, составляющие бюджета проекта, не оговоренные на предварительных этапах планирования проекта.

Для заказчика и исполнителя адекватная оценка стоимости проекта является очень важным фактором, который будет влиять на договор между ними. Добиться правильной оценки на предварительных стадиях разработки далеко не просто.

Целью любой разработки ИС состоит в удовлетворении потребностей заказчика. Соблюдать все требования технического задания и спецификации – это главная задача разработчика ПО. Очень редко можно встретить хорошо продуманную спецификацию на ИС, требования к системам часто по несколько раз возвращаются на доработки к заказчикам. В дополнение, многие IT-разработчики постоянно сталкиваются с проблемой «плывущих» требований. Конечно, IT-компании, работая по принципу «клиент прав», для удовлетворения требований заказчика идет на дополнительные

трудозатраты, но это создает массу проблем для проектировщиков и разработчиков, в первую очередь – риск не завершить проект в срок и даже его провалить.

Проанализировав причины изменения требований к разработке в процессе ее реализации, можно выделить следующие:

- Осознание заказчиком собственных потребностей. Иными словами, понимание того, что же ему на самом деле нужно.
- Смена бизнес–процессов и даже изменение бизнеса заказчика за время реализации проекта.

Указанные причины оказывают негативное влияние на процесс разработки, приводя к разногласиям между заказчиком и поставщиком и увеличению времени на создание ИС, влекущее превышение сроков. В итоге предварительная работа над проектом может оказаться сделанной впустую, будет превышен бюджет проекта и возникнут финансовые потери.

Оценка проекта – один из ключевых этапов разработки ИС и один из источников проблем проекта, если об этом забывают. Неправильная оценка особенно на предварительных этапах влечет серьезные ошибки на переходной стадии в проектирование. Попытаемся выделить факторы, влияющие на неправильную оценку:

- Незнание методик оценки проекта или отсутствие опыта. Это часто встречающаяся в IT-компаниях проблема и, наверное, самая существенная.
- Неправильная оценка рисков проекта, то есть непредвиденные затруднения в используемых средствах и компонентах.
- Ошибка аналитиков в оценке трудоёмкости.
- Недопонимание ключевых технических проблем проекта. Данная проблема, порой связана со сжатым графиком работ по проекту.
- Недостаток времени на изучение документации заказчика ПО.

На этапе планирования проекта на основе запрошенных работ осуществляется предварительная оценка, а именно оцениваются масштаб и атрибуты рабочих продуктов и задач. Для того чтобы установить границы планирования, строится модель жизненного цикла проекта. Затем производятся оценки трудозатрат и стоимости. Эти оценки используется в качестве основы для разработки проектных планов. Соблюдая рамки плана, устанавливаются бюджет и график проекта, выявляются проектные риски и создаются планы управления информацией и ресурсами, определяется потребность в знаниях и навыках, планируется привлечение к участию в проекте дополнительных участников. В заключительной стадии планирования существует острая необходимость привести запланированные ресурсы в соответствии с фактически имеющимися ресурсами.

Обычно оценки программных проектов представляются в виде обычных чисел, но подобные упрощенные точечные оценки бессмысленны, потому что они не включают никакой информации о вероятности, связанной с точечной оценкой (рис. 5.1). Подразумевается ситуация, когда возможен единственный исход – заданная точка.

Рис. 5.1 - Оценка программных проектов в виде обычных чисел

Выраженная таким образом оценка обычно оказывается целью, замаскированной под оценку.

Точные оценки учитывают, что программные проекты подвергаются влиянию множества факторов неопределенности (рис. 5.2). В совокупности эти факторы означают, что результат проекта подчиняется вероятностному распределению – одни результаты более вероятны, другие менее вероятны, а наиболее вероятная группа результатов сосредоточена где-то в середине распределения. Можно было бы предположить, что распределение результатов проекта будет иметь вид кривой нормального распределения.

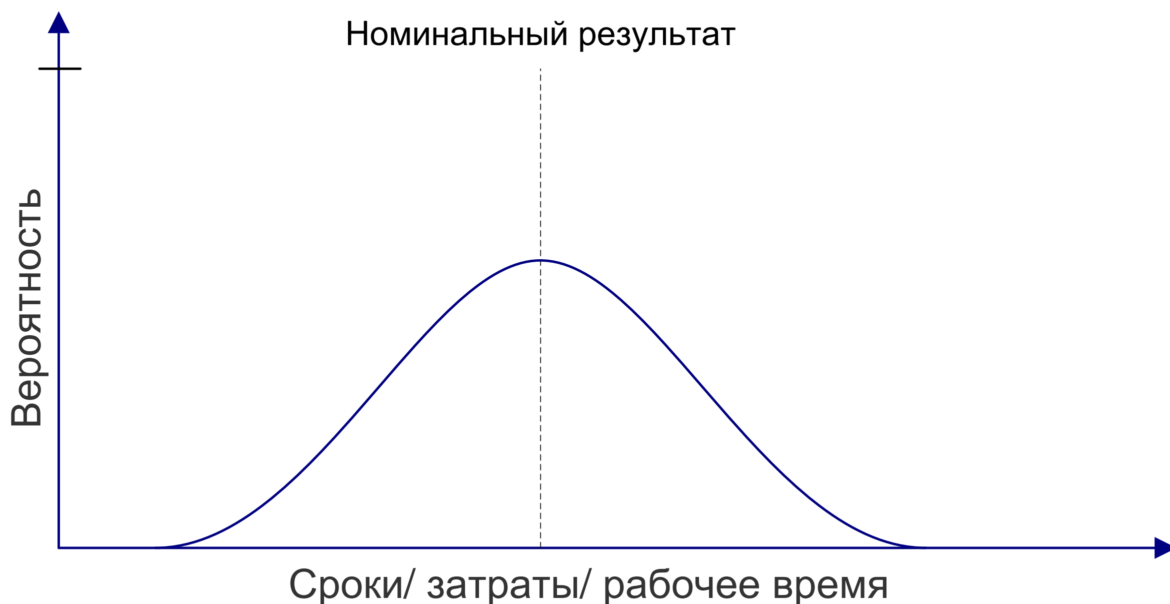

Рис. 5.2 - Оценка в виде распределение результатов проекта

Здесь каждая точка кривой представляет вероятность того, что проект завершится точно к соответствующей дате или на него уйдет определенная сумма. Площадь под кривой представляет суммарную вероятность 100%. Вероятностное распределение такого рода учитывает возможный разброс результатов. Тем не менее, предположение о симметричном распределении результатов по отношению к средней величине некорректно. Качество выполнения

проекта является величиной ограниченной: это означает, что левый край распределения усекается вместо того, чтобы бесконечно убывать, как в нормальном распределении (рис. 5.3). И хотя «удачные» результаты проекта ограничены, возможности для «неудач» безграничны, поэтому вероятностная кривая уходит очень далеко вправо.

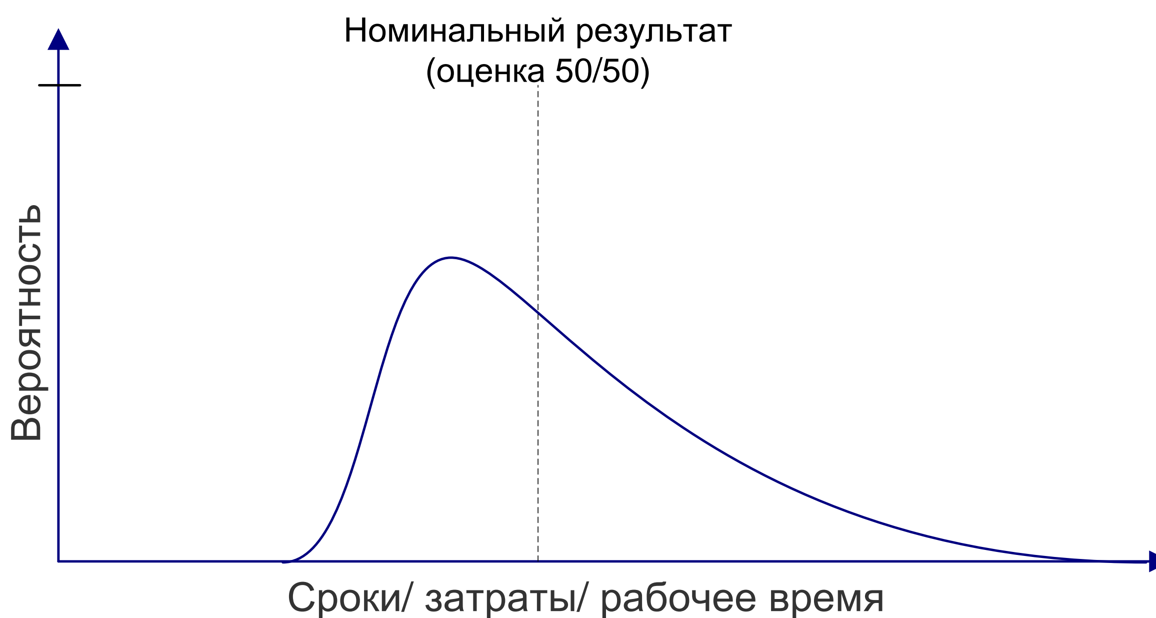

Рис. 5.3 - Несимметричная оценка

Основным назначением оценки программного проекта является вовсе не прогнозирование результата, а определение реалистичности целей проекта.

Проблемы начинаются тогда, когда разрыв между деловыми целями и сроками или объемом работ, необходимыми для достижения целей становится слишком большим. На практике показано, что если этот разрыв превышает 20% в ту или иную сторону, руководитель не сможет довести проект до успешного завершения, внося незначительные изменения за счет управления функциональностью, сроком, численностью группы и другими параметрами. Прежде чем руководитель начнет управление проектом для достижения

поставленных целей, эти цели необходимо привести в соответствие с реальностью.

Можно выявить ряд серьезных преимуществ, которые дает применение оценки программного проекта на различных стадиях:

1. Возможность отслеживания состояния проекта. Один из лучших способов – сравнение запланированного прогресса с фактическим. Если запланированный прогресс был основан на точных оценках и оказался достаточно реалистичным, то становится вполне возможным отслеживание прогресса на предмет соответствия планам.

2. Повышение качества. Точные оценки помогают избежать снижения качества, обусловленного приближающимся сроком сдачи. Как показали исследования, около 40% всех ошибок программирования возникает из-за стресса; этих ошибок можно было бы избежать за счет правильного планирования и снижения нагрузки на разработчиков. Также оказалось, что излишнее давление сроков служит основной причиной для выпуска модулей, содержащих ошибки, исправление которых обходится чрезвычайно дорого.

3. Улучшение координации с функциями, не связанными с программированием. Программные проекты обычно координируются с другими видами деятельности: тестированием, написанием документации, маркетинговыми кампаниями, обучением торгового персонала, финансовыми прогнозами. обучением службы поддержки и т.д. Ненадежный график проекта способен привести к сбоям взаимосвязанных функций. Хорошая оценка программного проекта предусматривает более тесную координацию всего проекта, включая и программные, и прочие виды деятельности.

4. Повышение качества бюджета. Точная оценка способствует выработке точного бюджета. Организация, не обеспечивающая точных оценок, подрывает свои возможности по прогнозированию стоимости проектов.

5. Получение ранней информации о рисках. Одной из самых частых упущенных возможностей в области программного обеспечения является неправильная интерпретация исходного несоответствия между целями и оценками проекта. Однако данное несоответствие несет чрезвычайно полезную информацию о риске, появившейся на ранней стадии проекта, когда еще возможны различные корректировочные меры: переопределение объема работ, набор дополнительного персонала, перевод лучших сотрудников на проект, изменение некоторых функций и т.д. Но если оставить данное несоответствие без внимания, возможностей корректировки на более поздней стадии будет гораздо меньше, и они будут обладать гораздо меньшей эффективностью. Как правило в таких случаях приходится выбирать между нарушением сроков и бюджета и отказом от важных функций.

6. Повышение доверия к группе разработчиков. Исполнительная группа, которая твердо держится на своих позициях и настаивает на точной оценке, пользуется большим доверием в своей организации.

5.2 Неопределенность в оценке

Разработка программного обеспечение состоит из множества решений относительно вопросов функциональности. Неопределенность в оценках программного обеспечения обусловлена разрешением неопределенности при принятии решений.

Исследователи обнаружили, что оценкам проектов на разных стадиях присущи прогнозируемые уровни неопределенности. Конус неопределенности на рисунке 5.4, показывает, что оценки становятся более точными по мере продвижения работы над проектом. Рассмотрим вначале конус неопределенности для последовательной методологии разработки.

Конус неопределенности для последовательного проекта

По горизонтальной оси отложены основные ключевые этапы проекта – исходная концепция, согласованное определение проекта, завершение постановки требований, завершение проектирование пользовательского интерфейса, завершение детального проектирования и, в итоге всего, готовый программный продукт. По вертикальной оси отсчитывается относительная величина ошибки в проектах, создаваемых опытными оценщиками на разных стадиях работы над проектом. Оценка может относиться к затратам или объему работы на реализацию определенного набора функций, количеству функций для заданного объема работы или срока и т.д. Вообще под объемом можно понимать размер проекта.

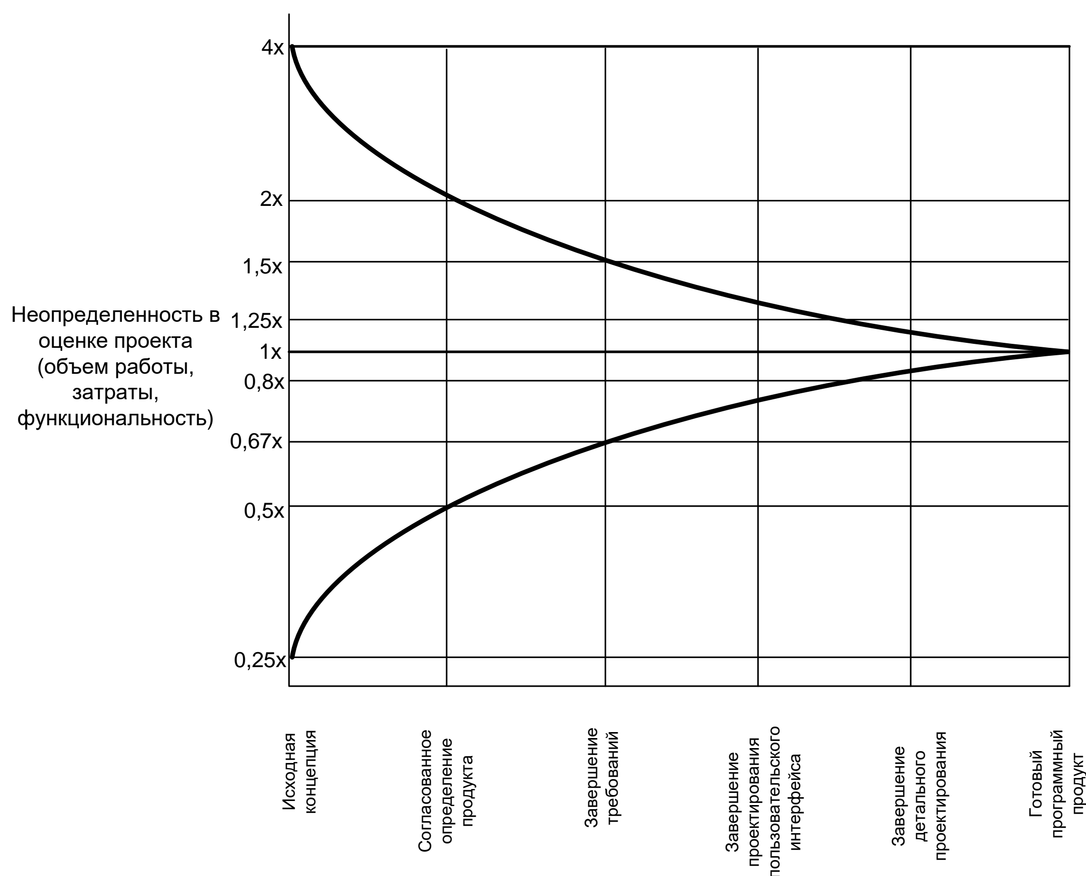

Рис. 5.4 - Конус неопределенности для последовательного проекта

Как видно из графика, оценки, создаваемые на очень ранней стадии проекта, подвержены высокой степени ошибок (рис. 5.4). Оценки, создаваемые на стадии исходной концепции, могут отличаться в большую или меньшую сторону до 4 раз. Соответственно, полный диапазон от верхней оценки до нижней составляет 4x/0,25x, то есть 16x.

Исследования показывают, что точность оценки программного проекта зависит от степени уточнения определения программы, то есть чем точнее определение, тем точнее оценка. Оценка изменчива, прежде всего, потому, что неопределенность заложена в самом проекте. И единственным способом сокращения неопределенности в оценке является сокращение ее в проекте.

Стандартное изображение конуса неопределенности создает ошибочное впечатление, будто конус сужается очень медленно – словно хорошая точность оценки становится возможной лишь тогда, когда работа над проектом почти завершена. Но это лишь иллюзия, возникающая из-за того, что ключевые точки на горизонтальной оси разделены равными интервалами.

В действительности все ключевые точки группируются в начальной части графика проекта. После перерисовки в календарном представлении конус принимает вид, показанный на рисунке ниже:

Как видно из этого рис. 5.5, точность оценки быстро возрастает в течение первых 30% проекта и улучшается с ± 4x до $\pm 1{,}25$x.

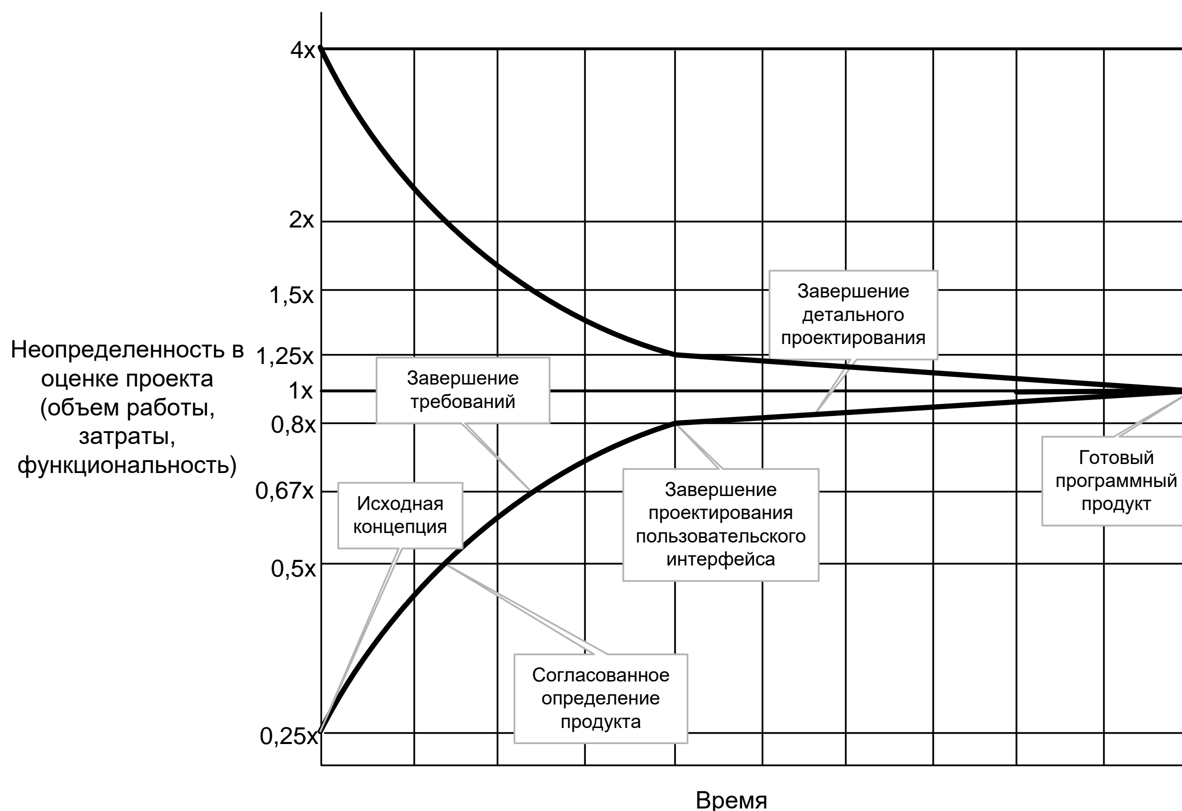

Рис. 5.5 - Точность оценки

Говоря о конусе неопределенности, необходимо учитывать одну важную концепцию: конус неопределенности представляет лучшую точность, которая может быть представлена в различных ключевых точках проекта, т.е. при недостаточно хорошем управлении проектом ожидаемого уточнения может и не быть. На рисунке 6. показано, что происходит, когда управление проектом не направлено на снижение неопределенности – последняя принимает вид не конуса, а облака, которое не рассеивается до самого конца проекта. Проблема даже не в том, что оценки не сходятся – не сходится сам проект.

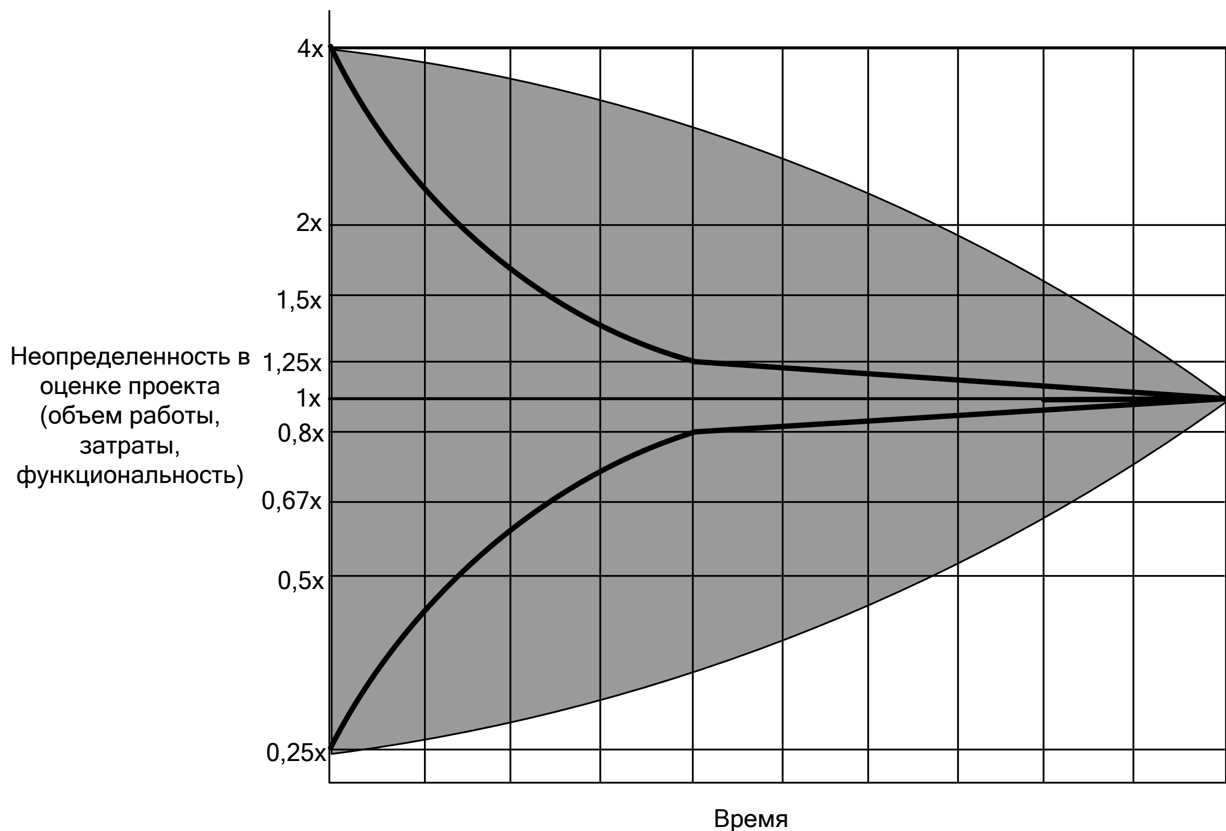

Рис. 5.6 - Облако неопределенности оценки проекта

Конус сужается только при принятии решений, направленных на устранение неопределенности: определение представлений о продукте, определение требований снижает неопределенность (рис. 5.6). Проектирование пользовательского интерфейса способствует снижению риска неопределенности, возникающего из-за неверного понимания требований.

Анализ оценок программных проектов показал, что специалисты, начинающие с точечных оценок и определяющие диапазоны на их основании, обычно не корректируют минимальное и максимальное значение с учетом неопределенности в оценке, особенно в ситуациях высокой неопределенности. Тенденция к использованию зауженных диапазонов преодолевается двумя способами.

Во-первых, можно начать с «наиболее вероятной» оценки, а затем вычислить диапазоны с использованием заранее определенных множителей как показано в табл. 5.1:

Таблица 5.1 - Ошибка оценки в ключевых точках работы над проектом

Фаза	**Ошибка**		
	Возможная ошибка в меньшую сторону	Возможная ошибка в большую сторону	Диапазон
Исходная концепция	0,25x (-75%)	4,0x (+300%)	16x
Согласованное определение продукта	0,50x (-50%)	2,0x (+100%)	4x
Завершение проектирования пользовательского интерфейса	0,67x (-33%)	1,5x (+50%)	2,25x
Завершение детального проектирования	0,90x (-10%)	1,10x (+10%)	1,2x

При использовании оценок из этой таблицы необходимо понимать, что в момент создания оценки мы еще не знаем, в какую сторону окажется смещенным фактический результат проекта – к началу или к концу диапазона.

Второй способ основан на отделении «оценки того, что мы знаем» от «оценки неопределенности». Один специалист дает оценки наилучшего и наихудшего случая – то есть концов диапазона, а другой оценивает вероятность того, что фактический результат войдет в этот диапазон.

Конус неопределенности для итеративного проекта

Учесть воздействие конуса неопределенности в итеративных проектах несколько сложнее, чем при традиционном последовательном подходе: если проект на каждой итерации проходит полный цикл разработки (от постановки требований до выхода готовой версии), то на каждой итерации возникает свой миниатюрный конус неопределенности. Перед постановкой требований для текущей итерации проект находится в точке согласованного определения продукта и подвержен 4-кратной амплитуде неопределенности в оценках. При коротких итерациях (меньше месяца) переход от согласованного определения проекта к фазам определения требований и завершения проектирования пользовательского интерфейса происходит за несколько дней, а неопределенность снижается с 4x до 1,6x. При фиксированном графике неопределенность 1,6x будет относиться к функциональности, готовой в отведенное время, а не к объему работ или срокам.

С другой стороны, при выборе подходов, при которых требования остаются неопределенными до начала каждой итерации, утрачивается возможность долгосрочного прогнозирования комбинаций затрат, сроков и функциональности, то есть их прогнозирование на несколько итераций спустя.

Многие группы разработчиков выбирают промежуточные методики, когда большинство требований определяется в начальной стадии работы над проектом, а проектирование, конструирование, тестирование и выпуск производятся короткими итерациями. Другими словами, проект движется последовательно до точки завершения проектирования пользовательского интерфейса (около 30% календарного времени), а затем переходит на более итеративный путь. Неопределенность, обусловленная воздействием конуса, снижается до $\pm 25\%$; это позволяет добиться поставленных целей при

качественном управлении проектом, не утрачивая основных преимуществ итеративной разработки. Промежуточный подход обеспечивает долгосрочную прогнозируемость затрат и сроков в сочетании с умеренной гибкостью в требованиях.

5.3 Факторы, влияющие на оценку

Факторы, оказывающие влияние на программный продукт, следует тщательно проанализировать, поскольку хорошее их знание повышает точность оценки и улучшает понимание общей динамики программного проекта.

Размер проекта

Важнейшим фактором влияния в оценке программного обеспечения является размер разрабатываемой программы, потому что он подвержен наибольшему разнообразию по сравнению со всеми остальными факторами.

На рис. 5.7 показана зависимость роста объема работ в среднем проекте бизнес-системы при увеличении размера проекта с 25 000 до 1 000 000 строк кода. Размер проекта на рисунке выражается в строках программного кода (LOC), но динамика остается неизменной независимо от того, в чем измеряется размер – в функциональных пунктах, длине списка требований, количестве веб-страниц или любых других показателях, выражающих те же диапазоны.

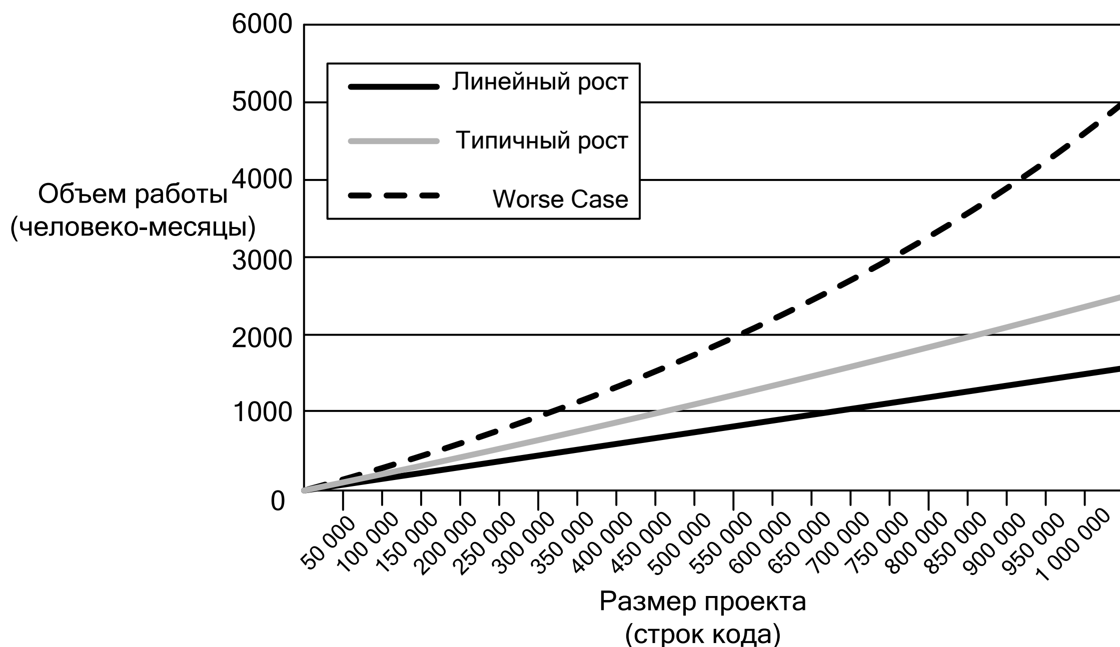

Рис. 5.7 - Динамика роста объема проекта

Как видно из диаграммы, система, состоящая из 1 000 000 строк кода потребует гораздо большего объема работы, чем система, состоящая из 100 000 строк. Казалось бы, на построение системы, в 10 раз большей другой, потребуется в 10 раз больше усилий. Однако объем работы для системы в 1 000 000 строк превышает 10-кратный объем работы для системы в 100 000 строк. Основная проблема заключается в том, что крупные проекты требуют координации между большим количеством групп, которым приходится общаться между собой. С ростом размера проекта число коммуникационных путей между людьми растет в квадратичной зависимости от количества участников проекта $(n \times (n-1))$. Следствием экспоненциального роста количества коммуникационных каналов является экспоненциальный рост трудоемкости с увеличением размера проекта. Данное явление называется издержками масштаба (diseconomy of scale). С увеличением масштаба системы стоимость каждой единицы повышается.

Также на рис. 5.7 продемонстрированы типичные издержки масштаба по сравнению с увеличением объема работы,

ассоциируемого с линейным ростом. Кроме номинальных издержек масштаба, на графике также показаны издержки в наихудшем случае. Из графика видно, что объем работы при худших издержках растет гораздо быстрее, чем при номинальных, а при больших размерах проекта эффект выражен гораздо ярче.

В издержках масштаба имеются как положительные, так и отрицательные стороны. Начнем с отрицательных: при существенных различиях в размере проектов новый проект нельзя оценивать применением простого масштабного коэффициента к объему работ, известному по предыдущим проектам. Скажем, если объем работ для предыдущего проекта в 100 000 строк кода составил 170 человеко-месяцев, можно предположить, что производительность составляет 100000/170, то есть 588 строк кода на человеко-месяц. Данное предположение может быть разумным для другого проекта примерно такого же размера, но если новый проект в 10 раз больше, такая оценка производительности может оказаться смещенной на величину от 30 до 200%.

Издержки масштаба можно смело игнорировать, когда новый проект по размеру мало отличается от прошлых проектов, поскольку при его оценке вполне можно использовать простой масштабный коэффициент – например, количество строк кода на человеко-месяц. И, как правило, большинство проектов, которыми занимается организация, имеют сходные размеры.

В описании графика в качестве единицы измерения использованы строки кода. Вообще существуют различные показатели размера программных проектов, в том числе:

- функции;
- требования;
- сценарии использования;
- функциональные пункты;
- web-страницы;

- компоненты GUI (окна, диалоговые окна, отчеты и т.д.);
- таблицы баз данных;
- определения интерфейсов;
- классы;
- строки программного кода.

Но стандартными метриками оценки размера, используемыми в моделях, являются строки программного кода и функциональные пункты. И те, и другие обладают разными достоинствами и недостатками и, как правило, создание оценок по разным метрикам и проверка их схождения или расхождения обеспечивает самые точные результаты.

На практике для оценки чаще всего используются именно строки программного кода (LOC). Такая оценка имеет как положительные, так и отрицательные стороны. С одной стороны, есть ряд преимуществ:

1. данные по количеству строк кода в прошлых проектах легко собираются при помощи служебных программ;
2. во многих организациях уже наработан большой объем исторических данных, выраженных в строках кода;
3. объем работы на строку кода остается более или менее постоянным для разных языков программирования;
4. измерения в строках кода позволяет выполнить межпроектные сравнения и оценивать будущие проекты по данным прошлых;
5. в большинстве коммерческих оценочных программ оценки объема работ и сроков в конечном счете основываются на строках кода.

С другой стороны, строки кода также создают определенные трудности при оценке размера:

1. упрощенные модели вида «количество строк кода на человеко-месяц» подвержены ошибкам из-за издержек масштаба и

заметных различий в скорости кодирования для разных типов программного обеспечения;

2. строки кода не могут использоваться в качестве основы для оценки задач, порученных отдельным программистам, из-за огромных различий в производительности;

3. если проект требует более сложного кода по сравнению с проектом, использовавшимся для калибровки предположений о производительности, это может нарушить точность оценки;

4. применение метрики LOC при оценке работы по постановке требований, проектированию и других действий, предшествующих созданию кода, выглядит противоестественно;

5. строки кода трудно оценивать напрямую и их обычно приходится оценивать опосредованно;

6. необходимо заранее тщательно определить, что именно следует считать строкой кода.

Многие эксперты возражают против использования строк кода в качестве метрики размера из-за проблем, возникающих при попытке анализа производительности в проектах с разными типами, размерами, языками программирования и предлагают использовать в качестве альтернативы LOC функциональные пункты. Это синтетическая метрика размера программы, которая может применяться для оценки размера проекта на ранних стадиях. Функциональные пункты проще вычислять по спецификации требований, чем строки кода; кроме того, они формируют основу для вычисления размера в строках кода. Существует много разных методов для вычисления функциональных пунктов. Стандарт подсчета функциональных пунктов поддерживается группой International Function Point Users Group (IFPUG).

Однако, проблемы, аналогичные проблемам использования строк кода, встречаются и при использовании других метрик размеров, включая функциональные пункты.

Тип программы

После размера проекта наибольшее влияние на оценку оказывает тип создаваемой программы. Если работа идет над критически важным проектом, к которому предъявляются особые требования, проект потребует гораздо большего объема работ, чем проект бизнес-системы аналогичного размера. В табл. 5.2 приведены примеры производительности (в строках кода на человеко-месяц) для проектов разных типов.

Таблица 5.2 - Производительность для стандартных типов проектов

Тип программы	**Строк кода на человеко-месяц (номинал)**		
	Проект на 10 000 строк кода	Проект на 100 000 строк кода	Проект на 250 000 строк кода
Авиационное оборудование	100-1000 (200)	20-300 (50)	20-200 (40)
Бизнес-система	800-18 000 (3000)	200-7000 (600)	100-5000 (500)
Системы управления	200-3000 (500)	50-600 (100)	40-500 (80)

Продолжение таблицы 5.2

Встроенные системы	100-2000 (300)	30-500 (70)	20-400 (60)
Интернет-системы (открытые)	600-10 000 (1500)	100-2000 (300)	100-1500 (200)
Интрасетевые системы (внутренние)	1500-18 000 (4000)	300-7000 (800)	200-5000 (600)
Микрокод	100-800 (200)	20-200 (40)	20-100 (30)
Управление процессами	500-5000 (1000)	100-1000 (300)	80-900 (200)
Системы реального времени	100-1500 (200)	20-300 (50)	20-300 (40)
Системы научных и инженерных исследований	500-7500 (1000)	100-1500 (300)	80-1000 (200)
Коммерческие пакеты	400-5000 (1000)	100-1000 (200)	70-800 (200)
Системные программы/ драйверы	200-5000 (600)	50-1000 (200)	40-800 (90)
Телекоммуникации	200-3000 (600)	50-600 (100)	40-500 (90)

Как видно из таблицы, группа, разрабатывающая интрасетевую систему для внутреннего использования, может генерировать код в 10-20 раз быстрее, чем группа, работающая над проектом управления авиационным оборудованием, системой реального времени или встроенной системой. Табл. 5.2, также в очередной раз демонстрирует издержки масштаба: в проектах на 100 000 строк код генерируется гораздо менее эффективно, чем в проектах на 10 000 строк, а в проектах на 250 000 строк эффективность оказывается еще ниже.

Область, для которой создается программное обеспечение, можно учесть тремя способами:

1. использовать результаты данной таблицы в качестве отправной точки;

2. использовать модель оценки и отрегулировать параметры оценки в соответствии со спецификой разрабатываемой программы;

3. использовать данные, полученные ранее организацией; тем самым в оценку автоматически включатся факторы разработки, действующие в отрасли.

Факторы персонала

Факторы, связанные с подбором, также оказывают значительное влияние на результат проекта (рис. 5.8). Так, например, в проекте на 100 000 строк кода суммарный эффект всех факторов персонала способен изменить оценку проекта в 22 раза.

Рис. 5.8 - Факторы персонала

Светлые полосы показывают увеличение объема работы по сравнению с номиналом в случае, если проект получает наихудший показатель в каждой категории, темные – уменьшение объема работы, если проект получает наилучший результат.

Воздействие этих факторов подтверждается многочисленными исследованиями, начавшимися еще в 1960-х годах и

демонстрировавшими различия от 10:1 до 20:1 в производительности отдельных участников и целых групп.

Одно из следствий подобных расхождений заключается в том, что точная оценка проекта невозможна без некоторого представления о том, кто будет заниматься его выполнением, потому что производительность работников может отличаться в 10 раз и более. Тем не менее в рамках одной конкретной организации такого разброса, скорее всего, не будет.

Язык программирования

Язык программирования, используемый в проекте, влияет на оценку, по меньшей мере, в четырех отношениях.

Во-первых, как видно из рисунка 5.8, опыт работы группы с конкретным языком и инструментарием, используемым в проекте способен изменить общую производительность в проекте до 40%.

Во-вторых, функциональность строки кода в разных языках программирования также не является постоянной величиной.

Третий фактор, также относящийся к языкам – широта возможностей инструментария и рабочих сред. Так слабый инструментарий и рабочая среда способны увеличить объем работы над проектом примерно на 50% по сравнению с сильными инструментариями и рабочими средами.

Последний фактор, связанный с языком программирования, заключается в том, что разработчики, использующие интерпретируемые языки, обычно работают продуктивнее тех, кто применяет компилируемые языки – выигрыш составляет до 2 раз.

5.4 Факторы, влияющие на выбор метода оценки

Обычно организации идут по одному из двух путей оценки проекта. Одни проекты начинаются с определения функциональности, а затем переходят к оценке сроков и объема

работы, необходимые для ее реализации. Другие проекты определяют свои бюджеты и временные рамки разработки, после чего выясняется, сколько функций можно реализовать за этот срок.

Многие методы оценки работают независимо оттого, что именно оценивается; некоторые методы лучше подходят для оценки объема работ, продолжительности или количества функций.

При выборе методики необходимо учитывать ряд факторов. Один из основных – размер проекта.

Размер проекта

К малым проектам (с пятью или менее техническими участниками) статистические методы обычно неприменимы, поскольку различия в производительности отдельных участников затмевают другие факторы. Как правило, в малых проектах используется плоская модель комплектования кадрами – на протяжении всего проекта численность персонала остается неизменной – из-за чего некоторые алгоритмические методы оценки, рассчитанные на большие проекты, становятся несостоятельными.

Лучшими методами оценки для малых проектов обычно оказываются «восходящие» методы, основанные на оценках людей, которые будут непосредственно заниматься выполнением работы.

К большим проектам относятся проекты, выполняемые группами около 25 участников, занимающие от 6 до 12 месяцев и более. Оптимальный выбор методики оценки для большого проекта существенно изменяется в зависимости от состояния проекта. На ранних стадиях лучший результат обычно дают «нисходящие» методы, основанные на алгоритмах и статистике. Они состоятельны на той стадии проекта, когда конкретный состав участников еще не известен. На средней стадии более точную оценку обеспечивает сочетание нисходящих и восходящих методов, базирующихся на

исторических данных самого проекта. На поздних стадиях крупных проектов восходящие методы дают наиболее точную оценку.

К категории средних проектов относятся проекты, выполняемые 5-25 участниками, занимающие от 3 до 12 месяцев. К преимуществам средних проектов можно отнести возможность применения практически всех методов оценки, применимых в крупных проектах, а также ряда методик малых проектов.

Стиль разработки

В контексте оценки выделяются два основных стиля разработки: последовательный и итеративный. Отраслевая терминология, окружающая итеративные, последовательные и динамические проекты, довольна сложна. Далее представлены некоторые подходы к разработке, основным различием между которыми является процент требований, определяемых на ранней стадии проекта, по сравнению с процентом требований, определяемых в ходе работы.

1. Эволюционное макетирование используется в тех случаях, когда требования неизвестны, а одна из главных причин для применения этой методики – содействие в определении требований. В контексте оценки эволюционное макетирование относится к итеративному стилю разработки.

2. Экстремальное программирование намеренно ограничиваться определением только тех требований, которые будут реализовываться при следующей итерации, обычно занимающей менее одного месяц. В контексте оценки относится к высокоитеративному стилю.

3. В проектах с эволюционной выдачей доля изначально определяемых требований изменяется от «почти отсутствует» до «большинства». В зависимости от того, к какому концу шкалы относится конкретный проект, он может быть как последовательным, так и итеративным. Как правило, проекты с эволюционной выдачей оставляют достаточно большое количество требований

неопределенными на момент начала разработки, чтобы разработку можно было отнести к итеративной.

4. В проектах с поэтапной выдачей основные требования определяются до начала основной работы над проектом. Поэтапная выдача использует итеративный подход к проектированию, конструированию и тестированию и поэтому в некотором смысле носит итеративный характер. Тем не менее в контексте оценки ее следует отнести к последовательному стилю разработки.

5. Стадии унифицированного процесса Rational (RUP) называются «итерациями», однако в типичном проекте RUP около 80% требований должны определяться до начала разработки. В контексте оценки RUP относится к последовательному стилю разработки.

6. Scrum – стиль разработки, при котором рабочая группа выбирает набор возможностей, которые она может реализовать в течение 30-дневного «броска». После того как «бросок» начался, клиенту не разрешается изменять требования. если рассматривать отдельные броски, в контексте оценки Scrum относится к последовательному стилю. Но поскольку функциональность не распределяется более чем по одному броску, с учетом множества итераций Scrum относится к итеративному стилю.

Как итеративные, так и последовательные проекты обычно начинаются с нисходящих, то есть основанных на статистике, методов и постепенно переходят к восходящим методам. Итеративные проекты гораздо быстрее уточняют свои оценки с использованием данных самого проекта.

Стадия разработки

По мере того, как группа работает над проектом, накапливается информация, позволяющая создать более точную оценку. Требования постепенно становятся более понятными, архитектура – более подробной, планы – более стабильными, а сам проект выдает данные

производительности, которые могут использоваться для оценки оставшейся части проекта.

Ранняя стадия. В последовательных проектах к ней относится период от начала построения концепции проекта и до того момента, когда требования в целом можно считать определенными. В итеративных проектах ранней стадией обозначается период исходного планирования.

Средней стадией называется период времени между начальным планированием и ранним конструированием. В последовательных проектах средняя стадия продолжается от этапа постановки требований и архитектуры до момента, когда проект будет в достаточной степени проработан для получения данных производительности, на основе которых может быть составлена оценка. В итеративных проектах под средней стадией понимаются первые две-четыре итерации, происходящие до того, как в основу оценок проекта можно будет уверенно заложить его собственные данные производительности.

Поздней стадией обычно обозначается время от середины разработки до выпуска.

Некоторые методы лучше всего работают в широкой части конуса неопределенности, другие обеспечивают лучшие результаты после того, как в ходе проекта будут сгенерированы данные, которые могут использоваться для оценки оставшейся части проекта.

Возможная точность

Точность методики зависит как от самой оценки, так и от того, насколько правильно выбрана методика для конкретной проблемы, и от специфики проекта.

Некоторые методы оценки обеспечивают высокую точность ценой высоких затрат. Другие обеспечивают более низкую точность при меньших затратах. Как правило, желательно использовать наиболее точные методы, однако в зависимости от текущего

состояния проекта и точности, возможной в текущей точке конуса неопределенности, метод с низкой точностью при низких затратах может оказаться даже более уместным.

Вопросы к главе 5

1. Каковы основные причины возникновения финансовых проблем в программных проектах?
2. Перечислите основные причины изменения требований в ходе реализации программного проекта.
3. Что такое оценка программного проекта?
4. Какой вид может иметь результат оценки программного проекта?
5. Что такое конус неопределенности?
6. Как меняется точность оценки в течение проекта?
7. Перечислите факторы, влияющие на оценку программного проекта.
8. Перечислите преимущества и недостатки использования числа строк кода при оценке программного проекта.
9. Как оценка проекта зависит от его типа?
10. Как оценка проекта зависит от персонала проекта?
11. Какие факторы влияют на выбор метода оценки программного проекта?
12. Каким образом размер проекта влияет на выбор метода оценки?
13. Каким образом стадия разработки влияет на выбор метода оценки?
14. От чего зависит точность оценки?

Глава 6. МЕТОДЫ ОЦЕНКИ СТОИМОСТИ ПО

6.1 Метод оценки по аналогии

Суть метода заключается в том, что для предсказания стоимости оцениваемого проекта используются фактические данные о стоимости прежде выполненных проектов. В основе этого метода лежит идея, что все проекты в чем-то схожи между собой.

Если сходство между проектом-аналогом и оцениваемым проектом велико, то результаты оценки могут быть очень точными, в противном случае оценка будет произведена неверно.

Пусть, например, требуется разработать новый программный продукт, и его модули аналогичны модулям другого, уже разработанного продукта, но должны содержать большее количество команд. По характеру работы предыдущий и предстоящий проекты очень схожи. Если объем работ в новом проекте на 30% больше, чем в предыдущем, то метод оценки «по аналогу» позволяет предположить, что и стоимость нового проекта будет на 30% больше стоимости предыдущего.

Базовый процесс оценки по аналогии выглядит следующим образом:

1. получить подробные данные об итоговом размере, объеме работ и затратах для предыдущего аналогичного проекта; если возможно, получить данные, фрагментированные по функциональности, структуре трудозатрат (WBS) или другой схеме декомпозиции;
2. шаг за шагом сравнить размер нового проекта с размером старого проекта;
3. построить оценку размера нового проекта в процентах от размера старого проекта;

4. создать оценку объема работ, руководствуясь размером нового проекта по сравнению с размером предыдущего проекта;

5. следить за тем, чтобы показатели старого и нового проектов базировались на единых предположениях.

Предположения необходимо проверять на каждом шаге. Впрочем, полноценная проверка некоторых предположений становится возможной только после завершения оценки. Перечислим основные источники рассогласования:

1. существенно различающиеся размеры старого и нового проекта, то есть различие более чем в 3 раза;

2. использование разных технологий (например, различные языки программирования);

3. существенные различия в квалификации отдельных участников в малых проектах или целых групп в больших проектах, небольшие различия вполне допустимы и часто просто неизбежны;

4. существенные различия в типе программы.

6.2 Метод параметрических оценок

Процесс оценки по параметру состоит в нахождении такого параметра проекта, изменение которого влечет пропорциональное изменение стоимости проекта. Математически параметрическая модель строится на основе одного или нескольких параметров. После ввода в модель значений параметров в результате расчетов получают оценку стоимости проекта.

Если параметрические модели различных проектов схожи и величину затрат и значения самих параметров легко подсчитать, то точность параметрической оценки предстоящего проекта можно повысить. Если, например, есть два выполненных проекта, причем стоимость одного из них больше стоимости оцениваемого проекта, а стоимость другого – меньше, и параметрическая модель справедлива

для обоих выполненных проектов, то точность параметрической оценки стоимости предстоящего проекта и надежность использования параметра будут достаточно высоки.

Оценку можно производить также с использованием множества параметров. В этом случае каждому параметру в зависимости от его значимости приписывается весовой коэффициент, и оценка стоимости осуществляется согласно многопараметрической модели.

6.3 Метод оценки «снизу вверх»

Метод оценки «снизу вверх» нужен для выработки согласованной базовой цены проекта или окончательной стоимостной оценки проекта. Название метода отражает способ расчета стоимостной оценки – метод предусматривает разбиение оценки на фрагменты, раздельную оценку каждого фрагменты и последующее объединение отдельных оценок в составную оценку стоимости всего проекта на более высоких уровнях обобщения. Такая методика также известна под названием «декомпозиции» и «восходящей оценки».

Разработка программного обеспечения представляет собой постепенное сокращение масштаба принимаемых решений.

Чем дальше продвигается проект к завершению, тем более детализированными становятся оценки, полученные в результате декомпозиции. На ранней стадии проекта восходящая оценка может базироваться на функциональных областях. Позднее за основу берется оценка требований. На завершающей стадии проекта можно использовать оценки уровня задач, предоставленные разработчиками и специалистами по тестированию.

Наиболее распространенным способом детализации проекта является декомпозиция проекта с применением структуры трудозатрат (Work Breakdown Structure – WBS). Она позволяет

избегать забытых задач в проекте и помогает сравнить оцениваемый проект с прошлыми.

В таблице 6.1 представлена обобщенная операционная структура WBS для программных проектов малого и среднего размера. Здесь перечислены операции и характер работы по каждой из них.

Таблица 6.1 - Обобщенная структура WBS

Категория	**Создание**	**Планирование**	**Управление**	**Обзор**	**Доработка**	**Сообщение о дефектах**
Общее управление	+	+	+	+		
Планирование	+		+	+	+	
Корпоративная деятельность	+					
Настройка конфигурации оборудования/ программного обеспечения/ сопровождение	+	+	+	+	+	+
Подготовка персонала	+	+	+	+		
Технические/ технологические процессы	+	+	+	+	+	+
Работа по требованиям	+	+	+	+	+	+
Координация с другими проектами	+					
Управление изменениями	+	+	+	+	+	+
Макетирование пользовательского интерфейса	+	+	+	+	+	+
Проработка архитектуры	+	+	+	+	+	+
Подробное проектирование	+	+	+	+	+	+
Программирование	+	+	+	+	+	+
Приобретение компонентов	+	+	+	+	+	+
Автоматизированная сборка	+	+	+	+	+	+
Интеграция	+	+	+	+	+	+
Ручное тестирование системы	+	+	+	+	+	+

Продолжение таблицы 6.1

Автоматизированное тестирование системы	+	+	+	+	+	+
Выпуск программы (внутренние, альфа- и бета-версии, окончательный выпуск)	+	+	+	+	+	+
Документация (пользовательская и техническая)	+	+	+	+	+	+

Обобщенная структура WBS создается посредством объединения столбцов с категориями. Самые распространенные комбинации отмечены в таблице плюсами.

Преимущество этого метода состоит в точности получаемых результатов, которая в свою очередь зависит от уровня детализации при оценке затрат на нижних уровнях рассмотрения. Из математической статистики известно, что чем больше деталей добавляется в рассмотрение, тем выше точность оценки.

Недостатком же этого метода является высокий уровень затрат средств и времени на выполнение детальной оценки.

6.4 Методы, основанные на экспертных оценках

Индивидуальные экспертные суждения до сих пор остаются самым распространенным методом оценки, применяемым на практике. Исследования показали, что порядка 80% оценщиков используют «экспертную оценку» проектов.

Индивидуальные экспертные оценки не обязаны быть неформальными или интуитивными. Между «интуитивными» и «структурированными» экспертными суждениями присутствуют значительные расхождения, поскольку первые оказывались в значительной степени неточными, а последние – не уступали по

точности оценкам, полученным при помощи математических моделей.

Для конкретных задач – таких, как время, необходимое на программирование и отладку некоторой функции, или создание набора тестовых сценариев – наиболее точные оценки создают люди, непосредственно выполняющие эту работу. Оценки людей, не связанных с работой, оказываются менее точными. Кроме того, сторонние оценщики склонны в большей степени к недооценкам, связанным с разработкой. Все это относится к оценкам уровня задач.

Итак, рассмотрим процесс создания экспертной оценки. Для начала, задачу необходимо разбить на несколько меньших, поскольку это один из лучших способов повышения точности. Далее оценщику (разработчику) предлагается оценить каждую из этих функций, для каждой из которых он выдает некоторое число – оценку. Затем оценщику нужно предположить выдать оценки для лучшего и худшего случая. В ходе анализа худшего случая иногда выявляется дополнительная работа, которая должна быть выполнена даже в лучшем случае, а это приводит к повышению номинальной оценки.

Создание оценок для лучшего и худшего случаев – всего лишь первый шаг. Далее стоит определить, какую же оценку использовать. Во многих случаях худший случай намного хуже того, что можно называть «ожидаемым случаем». Поэтому простое вычисление середин по диапазонам приведет к нежелательному смещению оценки. Для вычисления ожидаемого случая, который кстати может и не находиться в средней точке диапазона между лучшим и худшим случаями, разработана методика PERT (Program Evaluation and Review Technique). Для использования PERT в набор случаев включается дополнительный «наиболее вероятный» случай, который оценивается посредством экспертного суждения. Затем ожидаемый случай вычисляется по формуле:

$$ОжидаемыйСлучай = [ЛучшийСлучай + (4 \times НаиболееВероятныйСлучай) + ХудшийСлучай]/6$$

(1)

Формула (1) учитывает как полную длину диапазона, так и позицию наиболее вероятного случая внутри него.

Некоторые эксперты в области оценки рекомендуют изменить базовую формулу PERT, чтобы учесть смещение в оценке. Измененная формула выглядит так:

$$ОжидаемыйСлучай = [ЛучшийСлучай + (3 \times НаиболееВероятныйСлучай) + (2 \times ХудшийСлучай)]/6$$

(2)

Экспертные оценки могут быть не только индивидуальными, но и групповыми. Метод группового обсуждения полезен при оценке проекта на ранней стадии или при большом количестве факторов неопределенности. Рассмотрим два метода групповой экспертной оценки: неструктурированный и структурированный, называемый «широкополосным Дельфийским методом».

Групповое обсуждение. Простой способ повышения точности оценок, созданных отдельными экспертами, заключается в проведении группового анализа оценок. При обсуждении оценок в группах следует соблюдать три правила:

1. каждый участник группы оценивает фрагменты по отдельности, после чего группа встречается для сравнения оценок – стоит обсудить различия в оценках и понять причины различий, необходимо работать до тех пор, пока не будет достигнуто единое мнение по поводу верхней и нижней границ оценок;

2. не ограничиваться принятием усредненной оценки – вычислить среднее значение можно, но нужно обсуждать различия между отдельными результатами;

3. согласовать оценку, которая будет принята всей группой – не стоит прибегать к голосованию, если обсуждение заходит в тупик,

стоит обсудить различия и добиться консенсуса от всех участников группы.

Исследования показали, что средняя величина относительной ошибки для индивидуальных оценок составляет 55%. У оценок, прошедших обсуждение в группах, она уменьшается до 30%.

Широкополосный дельфийский метод относится к структурированным методам групповой оценки. Исходный Дельфийский метод был разработан в Rand Corporation в конце 1940-х годов для прогнозирования технологический тенденций. Его название происходит от древнегреческого города Дельфы, где находился оракул. В базовом варианте Дельфийского метода несколько экспертов создают независимые оценки, а затем встречаются до тех пор, пока им не удастся согласовать одну оценку.

Первоначальные исследования применения Дельфийского метода для оценки программного обеспечения покали, что базовый Дельфийский метод не обеспечивает большей точности, чем менее структурированные групповые обсуждения. Барри Боэм и его коллеги сделали вывод, что общие собрания слишком подвержены политическому давлению, и на них слишком часто доминируют наиболее настойчивые оценщики в группе. По этой причине Боэм с коллегами доработали базовый Дельфийский метод и превратили его в то, что сейчас известно под названием широкополосного Дельфийского метода.

Основная процедура заключается в следующем:

1. Координатор предоставляет каждому оценщику спецификацию и форму оценки.

2. Оценщики по отдельности готовят исходные оценки.

3. Координатор созывает собрание группы, на котором оценщики обсуждают проблемы, связанные с оценкой текущего проекта.

4. Оценщики анонимно передают индивидуальные оценки координатору.

5. Координатор готовит сводку оценок в итеративной форме и представляет форму оценщикам, чтобы они видели, как выглядят их оценки в сравнении с оценками других участников группы.

6. Координатор организует встречу, на которой оценщики обсуждают расхождения в оценках.

7. Оценщики проводят анонимное голосование по принятию средней оценки. Если хотя бы один из оценщиков проголосует отрицательно, процедура возвращается к шагу 3.

8. Окончательной считается точечная оценка, полученная по Дельфийской процедуре. Также окончательная оценка может представлять собой диапазон, созданный в ходе обсуждения, а точечная оценка представляет ожидаемый случай.

Рис. 6.1 - Итерации процесса оценивания

Все итерации оценок полезно отображать на одной шкале, чтобы разработчики могли проследить за их схождением (а в некоторых случаях – за расхождением) (рис. 6.1).

Исследования эффективности широкополосного Дельфийского метода показали, что он снижает ошибку примерно на 40% по сравнению с усредненной оценкой группы. Наиболее полезным он является при оценке работы, проводимой в малознакомой деловой

области, базирующейся на новой технологии или направленной на разработку нового вида программного обеспечения. Он помогает создавать приближенные оценки «с точностью до порядка» на стадии определения продукта или выработки концепции, пока многие требования еще не были сформулированы. Он также хорошо работает в проектах, связанных с несколькими разнородными специализациями. Кроме того, широкополосный Дельфийский метод способствует более четкому определению объема работы и помогает избавиться от лишних предположений. Таким образом, наибольшую пользу он приносит при оценке одиночных показателей, использующих входные данные из разных областей в очень широкой части конуса неопределенности.

Но для множества задач для получения подробных оценок метод не подходит, поскольку требует проведения собраний, а следовательно больших затрат рабочего времени. Поэтому главным недостатком широкополосного Дельфийского метода считается его дороговизна.

6.5 Линейный метод

В простейшем случае определить стоимость разработки ПО можно, исходя из количественной оценки трудозатрат (в неких единицах, например, человеко-месяцах или человеко-часах) и их удельной стоимости:

$$C = T \times Ц \qquad (3)$$

Цена одной единицы трудозатрат для индустрии разработки ПО формируется, в основном, исходя из заработной платы и связанных с ней начислений. Как правило, другие составляющие имеют гораздо меньший удельный вес, и ими зачастую можно пренебречь.

Что касается самих трудозатрат, то их достоверное вычисление в сфере интеллектуальной деятельности, к которой, несомненно, относится разработка ПО, выполнить достаточно сложно. Простейший подход может быть основан на следующей линейной формуле:

$$T = P \times \Pi \qquad (4)$$

где Р – размер исходного кода ПО;
П – временная производительность.

Эта примитивная формула активно применяется и сейчас, хотя ее несостоятельность была установлена довольно давно. Пожалуй, самой известной работой, в которой критикуется данный подход, является выдержавшая более двадцати изданий по-настоящему классическая книга Фредерика Брукса «Мифический человеко-месяц, или Как создаются программные системы», впервые увидевшая свет еще в 1977 г.

По мнению Брукса, методы оценки ошибочно путают достигнутый прогресс с затраченными усилиями. В первую очередь, это касается способа, которым измеряется результат, – на заре программирования не было найдено ничего лучшего, чем использование в этих целях количества строк кода. С ростом мастерства программист обычно делается «лаконичнее», т. е. выдаваемый им код для решения одних и тех же задач становится все компактнее, а это означает, что его проще и дешевле отлаживать и сопровождать. Однако вышеуказанная формула вовсе не стимулирует данного процесса.

6.6 Методы, основанные на функциональных пунктах

Наверное, наиболее удачной заменой количеству строк кода для измерения производительности стали функциональные пункты (function points), впервые предложенные сотрудником IBM Аланом Альбрехтом в 1979г. Наиболее важное преимущество данного метода: поскольку применение функциональных пунктов основано на изучении требований, то оценка необходимых трудозатрат может быть выполнена на самых ранних стадиях работы над проектом и далее будет уточняться по ходу жизненного цикла, а явная связь между требованиями к создаваемой системе и получаемой оценкой позволяет заказчику понять, за что именно он платит, и во что выльется изменение первоначального задания.

Постепенно метод функциональных пунктов превратился в индустриальный стандарт, и в 1986 г. для его поддержки и развития была создана некоммерческая организация IFPUG (International Function Point User Group). Кроме того, он послужил основой для множества производных подходов.

Точки свойств

В условиях, когда сформулированные требования не отражают истинной сложности реализации (что особенно характерно для системного ПО, критически важных программных комплексов и пр.), метод функциональных пунктов себя не оправдывает. В этом случае на помощь приходит его модифицированный вариант, предложенный в 1988 г. Кейперсом Джонсом, который учитывает не только требования к системе, но и внутренние особенности ее реализации – метод точек свойств (feature points). Он очень близок к методу функциональных пунктов, с тем лишь отличием, что предусматривает корректирование получаемой оценки с учетом алгоритмической сложности.

Метод Mark II

Еще одна примечательная модификация метода функциональных пунктов, представленная Чарльзом Саймонсом также в 1988 г. Автор стремился избавиться от многих известных его недостатков и сделать более пригодным для оценки сложных систем. В частности, Mark II позволяет добиться одного и того же результата как при оценке системы в целом, так и при суммировании оценок, полученных для составляющих ее подсистем.

Трехмерные функциональные пункты

Еще одно логическое развитие оригинального подхода было предложено софтверным подразделением корпорации Boeing в 1991 г. В основу этого метода положена идея о том, что сложность задачи в программной среде можно представить в трех измерениях – данные (количество вводов/выводов), функции (сложность вычислений) и контроль (управляющая логика). Важно отметить, что он выходит за рамки исключительно программных проектов и позволяет оценивать трудоемкость решения задач в различных сферах – деловой, научной и т. д.

Объектные пункты

Поскольку классическая интерпретация метода функциональных пунктов не предусматривает применения объектно-ориентированного подхода, в современных проектах используется его адаптированный вариант, оперирующий именно терминами объектно-ориентированной технологии.

Подсчет функциональных пунктов – метод IFPUG

Методика анализа функциональных пунктов основывается на концепции разграничения взаимодействия (рис. 6.2). Сущность ее состоит в том, что программа разделяется на классы компонентов по формату и типу логических операций. В основе этого деления лежит предположение, что область взаимодействия программы разделяется

на внутреннюю – взаимодействие компонентов приложения, и внешнюю – взаимодействие с другими приложениями.

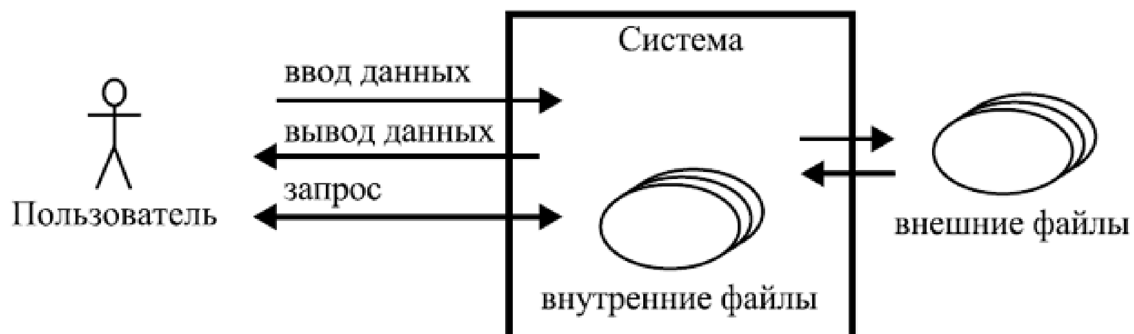

Рис. 6.2 - Схема рассмотрения системы при оценке ее сложности в функциональных точках показана на рисунке

В соответствии с принятым стандартом размер программы в функциональных пунктах базируется на количестве и сложности следующих элементов:

1. Внешние входные элементы External Inputs (EI) – экраны, формы, диалоговые окна или управляющие сигналы, при помощи которых пользователь или внешняя программа добавляет, удаляет или изменяет данные программы. К этой категории относятся все входные элементы, обладающие уникальным форматом или уникальной логикой обработки.

2. Внешние выходные элементы External Outputs (EO) – экраны, отчеты, диаграммы или управляющие сигналы, генерируемые программой для пользователя или внешних программ. К этой категории относятся все выходные элементы, отличающиеся по формату или логике обработки от других типов вывода.

3. Внешние запросы External Inquiry (EQ) – комбинации входных/выходных элементов, в которых входному элементу ставится в соответствие простая выходная форма. Термин происходит из мира баз данных и относится к прямому поиску данных. В современных графических и web-приложениях граница между запросами и выходными элементами размыта, но в общем случае

запросы производят выборку данных непосредственно из базы и ограничиваются минимальным форматированием, а выходные элементы поддерживают обработку, комбинирование и обобщение сложных данных с широкими возможностями форматирования.

4. Внутренние логические файлы Internal Logical Files (ILF's) – основные логические группы пользовательских или управляющих данных находящихся под полным контролем программы. Логический файл представляет собой один неструктурированный файл или одну таблицу в реляционной базе данных.

5. Внешние интерфейсные файлы External Interface Files (EIF's) – файлы, находящиеся под контролем других программ, с которыми взаимодействует измеряемая программа. К этой категории относятся все основные группы логических и управляющих данных, принимаемых или передаваемых программой.

Следующим шагом после распределения требований спецификации по пяти главным классам является оценка их сложности. Классы компонентов оцениваются по сложности и относятся к категории высокого, среднего или низкого уровней сложности. Для транзакций (EI, EO, EQ) уровень определяется по количеству файлов, на которые ссылается транзакция File Types Referenced (FTR) и количеству типов элементов данных Data Element Types (DET). Для ILF и EIF имеют значение типы элементов записей Record Element Types (RET) и DET. Типы элементов записей - это подгруппа элементов данных в ILF или EIF. Типы элементов данных – это уникальное не рекурсивное поле подмножества ILF или EIF.

Таблица 6.2 - Внешние входы

Ссылки на внутренние логические файлы (FTR)	**Элементы данных (DET)**		
	1–4	5–15	>15
0–1	Низкий	Низкий	Средний
2	Низкий	Средний	Высокий
>2	Средний	Высокий	Высокий

Таблица 6.3 - Внешние выходы

Ссылки на внутренние логические файлы (FTR)	**Элементы данных (DET)**		
	1–4	5–19	>19
0–1	Низкий	Низкий	Средний
2–3	Низкий	Средний	Высокий
>3	Средний	Высокий	Высокий

Таблица 6.4 - Внешние запросы

Ссылки на внутренние логические файлы (FTR)	**Элементы данных (DET)**		
	1–4	5–19	>19
0–1	Низкий	Низкий	Средний
2–3	Низкий	Средний	Высокий
>3	Средний	Высокий	Высокий

Таблица 6.5 - Внутренние логические файлы

Разнообразие записей (RET)	**Элементы данных (DET)**		
	1–19	20–50	>50
1	Низкий	Низкий	Средний
2–5	Низкий	Средний	Высокий
>5	Средний	Высокий	Высокий

Таблица 6.6 - Внешние интерфейсные файлы

Разнообразие записей (RET)	**Элементы данных (DET)**		
	1–19	20–50	>50
1	Низкий	Низкий	Средний
2–5	Низкий	Средний	Высокий
>5	Средний	Высокий	Высокий

В табл. 6.7 показано как счетчики входных элементов, выходных элементов и т.д. преобразуются в некорректируемые функциональные пункты.

Таблица 6.7 - Множители для вычисления нескорректированных функциональных пунктов

Характеристика программы	Функциональные пункты		
	Низкая сложность	Средняя сложность	Высокая сложность
Внешние входные элементы	$\times 3$	$\times 4$	$\times 6$
Внешние выходные элементы	$\times 4$	$\times 5$	$\times 7$
Внешние запросы	$\times 3$	$\times 4$	$\times 6$
Внутренние логические файлы	$\times 7$	$\times 10$	$\times 15$
Внешние интерфейсные файлы	$\times 5$	$\times 7$	$\times 10$

После суммы нескорректированных функциональных пунктов вычисляется коэффициент влияния; он основывается на влиянии, оказываемом на программу 14 факторами.

1. Связи данных. Как много связей используются для передачи данных внутри системы.
2. Распределенная обработка данных. Как обрабатываются распределенные данные.
3. Какие требования по производительности.
4. Какие требования по аппаратной части.
5. Как часто выполняются транзакции (ежедневно, еженедельно)
6. Какой процент информации вводится онлайн.
7. Эффективность для конечного пользователя.
8. Как много ILF обновляются онлайн.
9. Используются ли сложная логическая или математическая обработка.
10. Была ли спроектирована программа для одного или многих пользователей. Насколько сложна инсталляция.

11. Насколько эффективны или автоматизированы старт, резервное копирование и восстановление.

12. Было ли приложение спроектировано, разработано для множества сотрудников множества организаций.

13. Было ли приложение спроектировано для упрощения дальнейших изменений в нем.

Степень влияния каждого фактора оценивается от 0 до 5, то есть от отсутствия какого-либо влияния до сильного влияния. Коэффициент влияния лежит в диапазоне от 0,65 до 1,35 и вычисляется по формуле:

$$0{,}65 + \frac{\sum C_i}{100}, \qquad (5)$$

где C_i – влияние i-того фактора.

После умножения нескорректированной суммы на коэффициент влияния получается скорректированная величина в функциональных пунктах.

Итак, получается, что количество функциональных пунктов вычисляется по формуле:

$$FP = UAF \times VAF \qquad (6)$$

где FP – конечное количество функциональных пунктов,
VAF – коэффициент влияния,
UAF – количество нескорректированных функциональных пунктов и

$$UAF = EI + EO + EQ + ILF + EIF \qquad (7)$$

Если необходимо учесть предварительную обработку данных или дополнительные требования, то используют следующие формулы для расчета:

Учет разработки с предварительной обработкой данных

$$DFP = (UFP + CFP) \times VAF \tag{8}$$

где DFP - количество функциональных пунктов разработки,
UFP - количество нескорректированных функциональных пунктов,
CFP - количество функциональных пунктов добавленных для обработки прочих функциональных пунктов,
VAF – коэффициент влияния.

Учет дополнительных требований:

$$EFP = \left[(ADD + CHGA + CFP) \times VAFA\right] + (DEL \times VAFB) \tag{9}$$

где EFP - количество дополнительных функциональных пунктов,
ADD - количество нескорректированных функциональных пунктов, которые были добавлены по дополнительным требованиям,
CHGA - количество нескорректированных функциональных пунктов, которые были изменены по дополнительным требованиям,
CFP - количество нескорректированных функциональных пунктов, которые были добавлены после преобразований, обычно CFP = 0,
VAFA - коэффициент влияния после дополнительных требований,
VAFB - коэффициент влияния до дополнительных требований,
DEL - количество нескорректированных функциональных пунктов, которые были удалены после преобразований.

Сама по себе оценка функциональности в баллах иногда не подходит для того, чтобы определить сложность проекта и его

стоимость. Поэтому предусмотрен переход от функциональных пунктов к строкам кода при помощи коэффициентов преобразования (табл. 6.8).

Таблица 6.8 - Коэффициенты перехода

Язык программирования	Среднее значение	Нижнее значение	Верхнее значение
Access	35	15	47
Ada	154	104	205
Advantage	38	38	38
APS	86	20	184
ASP	69	32	127
Assembler	172	86	320
C	148	9	704
C++	60	29	178
C#	59	51	66
Clipper	38	27	70
COBOL	73	8	400
Cool:Gen/IEF	38	10	180
Culprit	51	-	-
DBase IV	52	-	-
Easytrieve+	33	25	41
Excel	47	31	63
Focus	43	32	56
FORTRAN	-	-	-
FoxPro	32	25	35
HTML	43	35	53
Ideal	66	34	203
IEF/Cool:Gen	38	10	180
Informix	42	24	57
J2EE	61	50	100
Java	60	14	97
JavaScript	56	44	65
JCL	60	21	115
JSP	59	-	-
Lotus Notes	21	15	25

Mantis	71	22	250
Mapper	118	16	245
Natural	60	22	141
Oracle	38	4	122
Oracle Dev 2K/FORMS	41/42	21/23	100
Pacbase	44	26	60
PeopleSoft	33	30	40
Perl	60	-	-
PL/1	59	22	92
PL/SQL	46	14	110
Powerbuilder	24	105	-
REXX	67	-	-
RPG II/III	61	24	155
Sabretalk	80	54	99
SAS	40	33	49
Siebel Tools	13	5	20
Slogan	81	66	100
Smalltalk	35	17	55
SQL	39	15	143
VBScript	45	27	50
Visual Basic	42	276	-
VPF	96	92	101
Web Scripts	44	9	114

Упрощенный метод вычисления функциональных пунктов – элементы GUI

При вычислении функциональных пунктов необходимо строку за строкой перебрать спецификацию требований и буквально подсчитать все входные и выходные элементы, файлы и т.д. На это может потребоваться много времени.

Эксперты в области оценки предложили упрощенный метод вычисления функциональных пунктов, основанный на подсчете элементов графического интерфейса (GUI): экраны, формы и отчеты, присутствующие в системе, классы, а также модули, написанные на необъектных языках – объектных пунктов. Сложность каждого из

таких элементов оценивается отдельно, после чего их сложности складываются, тоже с разными весовыми коэффициентами для разных категорий элементов. В общем случае процесс состоит из следующих шагов:

1. подсчитать количество элементов GUI по категориям из табл. 6.9:

Таблица 6.9 - Элементы GUI

Элемент GUI	Эквивалент в функциональных пунктах
Простое клиентское окно	Один внешний входной элемент низкой сложности для операций добавления, изменения и удаления + один внешний запрос низкой сложности
Среднее клиентское окно	Один внешний входной элемент средней сложности для операций добавления, изменения и удаления + один внешний запрос средней сложности
Сложное клиентское окно	Один внешний входной элемент высокой сложности для операций добавления, изменения и удаления + один внешний запрос высокой сложности
Средний отчет	Один внешний входной элемент средней сложности
Сложный отчет	Один внешний входной элемент высокой сложности
Любой файл	Один внутренний логический файл низкой сложности
Простой интерфейс	Один внешний входной элемент низкой сложности для получения данных + один внешний входной элемент низкой сложности для выдачи данных
Средний интерфейс	Один внешний входной элемент средней сложности для получения данных + один внешний входной элемент средней сложности для выдачи данных
Сложный интерфейс	Один внешний входной элемент высокой сложности для получения данных + один внешний входной элемент высокой сложности для выдачи данных
Окно сообщения или диалоговое окно	Не учитываются по отдельности – только в составе экрана, с которым они связаны

2. преобразовать количество элементов GUI в количество функциональных пунктов;

3. вычислить размер в строках кода.

При использовании этого подхода необходимо понимать, какая неопределенность закладывается в оценку. Поскольку преобразование элементов GUI в функциональные пункты и последующее преобразование их в строки кода вводит большую долю неопределенности в оценку.

Оба метода – расчет по функциональным и объектным пунктам, хорошо применимы к так называемым информационным системам, т.е. системам, основные функции которых связаны с накоплением и хранением больших объемов данных, а также с предоставлением доступа и интерактивной обработкой запросов к ней.

Метод ISBSG

Группа ISBSG (International Software Benchmarking Standard Group) разработала методику вычисления объема работ, основанную на трех факторах: размере программного проекта в функциональных пунктах, типа среды разработки и максимальном размере группы. Для разных типов проектов разработаны соответствующие формулы, которые выдают оценку в человеко-месяцах в предположении, что один человеко-месяц составляет 132 часа плотной работы над проектом (то есть за исключением отпусков, выходных, обучения, собраний и т.д.).

Тип: общий

$$ЧеловекоМесяцы = 0{,}512 \times ФункциональныеПункты^{0{,}392} \times МаксимальныйРазмерГруппы^{0{,}791} \quad (10)$$

Тип: проект для мейнфреймов

$$ЧеловекоМесяцы = 0{,}685 \times ФункциональныеПункты^{0{,}507} \times МаксимальныйРазмерГруппы^{0{,}464} \quad (11)$$

Тип: проект среднего диапазона

$$ЧеловекоМесяцы = 0{,}472 \times ФункциональныеПункты^{0{,}375} \times МаксимальныйРазмерГруппы^{0{,}882}$$
(12)

Тип: настольная система

$$ЧеловекоМесяцы = 0{,}157 \times ФункциональныеПункты^{0{,}591} \times МаксимальныйРазмерГруппы^{0{,}810}$$
(13)

Тип: языки третьего поколения

$$ЧеловекоМесяцы = 0{,}425 \times ФункциональныеПункты^{0{,}488} \times МаксимальныйРазмерГруппы^{0{,}697}$$
(14)

Тип: языки четвертого поколения

$$ЧеловекоМесяцы = 0{,}317 \times ФункциональныеПункты^{0{,}472} \times МаксимальныйРазмерГруппы^{0{,}784}$$
(15)

Тип: доработка существующих проектов

$$ЧеловекоМесяцы = 0{,}669 \times ФункциональныеПункты^{0{,}338} \times МаксимальныйРазмерГруппы^{0{,}758}$$
(16)

Тип: новая разработка

$$ЧеловекоМесяцы = 0{,}520 \times ФункциональныеПункты^{0{,}385} \times МаксимальныйРазмерГруппы^{0{,}866}$$
(17)

Интересная особенность метода ISBSG состоит в том, что формулы для объема работ зависят от максимального размера группы, а команды меньшего размера уменьшают общий объем. С точки зрения оценки это создает неопределенность. С точки же

зрения управления проектом эти изменения могут заставить использовать группу меньшего размера вместо большей.

Методики подсчета функциональных пунктов могут быть успешно применены для:

- определения трудоемкости и стоимости планируемых проектов разработки программного обеспечения;
- проведения сравнительного анализа качества и производительности разработки разнотипных проектов, или однотипных проектов, при выполнении которых использовались различные технологии,
- проведения анализа плановой и реальной оценки сложности и величины разработанного программного обеспечения и трудоемкости выполнения проекта, получения стандартной метрики сравнения программных продуктов.

Несмотря на преимущества методов оценки по функциональным пунктам они имеют ряд недостатков, существенно ограничивающих возможности применения. Ниже перечислены основные из них:

- для работы по методу функциональных пунктов требуются высококвалифицированные сертифицированные специалисты;
- расчет количества функциональных пунктов требует значительного времени на сбор данных о системе и получение полного понимания требований к системе;
- при применении метода на ранних этапах разработки многократно понижается точность оценки;
- метод «интуитивно» не понятен, так как оперирует многими понятиями, неиспользуемыми в современном объектно-ориентированном подходе.

Существуют приложения, в оценке которых использование стандартных функциональных пунктов не эффективно:

- управление процессом в реальном времени;
- математические вычисления;

- симуляция;
- системные приложения;
- инженерные приложения;
- встроенные системы.

Перечисленные приложения отличаются высокой интенсивностью вычислений, часто основанных на алгоритмах повышенной сложности. Для решения задач расчёта размера указанных приложений в 1986 году организацией Software Productivity Research (SPR) была разработана методика анализа характеристических пунктов ПО (feature points). Сущность ее состоит в том, что оценивается количество алгоритмов в программе и незначительно модифицируется степень значимости для расчёта функциональных пунктов. Эта методика считается экспериментальной.

6.7 Методы COCOMO и COCOMO II

Пожалуй, самой популярной моделью для оценки стоимости разработки ПО, которая де-факто стала стандартом, является COCOMO (COnstructive COst MOdel). Она была представлена в 1981 г. Барри Боэмом, известным ученым, внесшим огромный вклад в развитие научных подходов к управлению программными проектами – им разработаны спиральная модель проектирования ПО и Wideband Delphi, кроме того, когда-то именно он предсказал, что в будущем стоимость ПО превысит стоимость оборудования.

COCOMO создана на основе анализа статистических данных 63 проектов различных типов. Фактически под общим названием скрываются три уровня детализации: базовый, промежуточный и подробный. Также предусмотрено три режима использования модели в зависимости от размеров команды и проекта (табл. 6.10).

Таблица 6.10 - Режимы модели COCOMO

Название режима	**Размер проекта**	**Описание**
Органичный	До 50 KLOC	Некрупный проект разрабатывается небольшой командой, для которой нехарактерны нововведения, и среда остается стабильной
Сблокированный	50–300 KLOC	Относительно небольшая команда занимается проектом среднего размера, в процессе разработки необходимы определенные инновации, среда характеризуется незначительной нестабильностью
Внедренный	Более 300 KLOC	Большая команда разработчиков трудится над крупным проектом, необходим значительный объем инноваций, среда состоит из множества элементов, которые не характеризуются стабильностью

Для оценки трудозатрат на базовом уровне модели COCOMO применяется следующая формула:

$$T = a \times Pb \qquad (18)$$

где a и b – константы, которые зависят от режима использования модели.

В соответствии с этой формулой трудозатраты вообще нелинейно зависят от размера проекта и скачкообразно изменяются при смене режима (табл. 6.11). Другая интересная особенность COCOMO – рост трудозатрат при переходе к более высокому режиму не означает безусловного увеличения длительности (F) выполнения проекта, которая вычисляется по формуле:

$$F = 2.5 \times Tk \qquad (19)$$

поскольку при этом изменяется значение константы k.

Таблица 6.11 - Значения коэффициентов модели COCOMO в зависимости от режима использования

Название режима	Значение коэффициента a	Значение коэффициента b	Значение коэффициента k
Органичный	2,4	1,05	0,38
Сблокированный	3,0	1,12	0,35
Внедренный	3,6	1,20	0,32

На более высоких уровнях COCOMO рассмотренные формулы усложняются, они обрастают дополнительными коэффициентами, позволяющими повысить точность оценок. Также модель допускает калибровку на основе хронологических данных по выполненным проектам.

COCOMO II

Сегодня оригинальная COCOMO уже считается устаревшей, ей на смену пришла COCOMO II, представленная в 1997 г. Хотя она и имеет много общего со своей предшественницей, однако во многом основана на новых идеях, а также адаптирована к современным методологиям разработки ПО (в частности, если COCOMO подразумевала только каскадную модель жизненного цикла, то COCOMO II также пригодна для спиральной и итеративной).

При построении COCOMO II для обработки статистических данных использовался Байесовский анализ, который дает лучшие результаты для программных проектов, характеризующихся неполнотой и неоднозначностью, в отличие от многофакторного регрессионного, примененного в COCOMO. Также в ней допускается измерять размер проекта не только числом строк кода, но и более современными функциональными и объектными точками. Помимо прочего, при расчете показателей COCOMO II учитывает уровень

зрелости процесса разработки в соответствии с моделями SEI CMM/CMMI.

Как и COCOMO, COCOMO II также имеет несколько вариантов использования, однако они отличаются не столько детализацией, сколько характером – фактически это разные модели для решения разных (хотя и схожих) задач, объединенные под одним общим названием (таблица 6.12). При этом формулы для вычисления различных показателей значительно усложнились, и мы не будем их здесь приводить, отметим лишь, что при сохранении основных принципов модель стала намного гибче и учитывает гораздо большее число факторов, влияющих на выполнение программного проекта.

Таблица 6.12 - Модель COCOMO II фактически объединяет три различные подмодели

Название модели	**Описание**
Композиционная прикладная	Ориентирована на проекты, создаваемые с применением современных инструментальных средств и UML, использует в качестве метрики объектные точки
Ранней разработки проекта	Применяется для получения приближенных оценок по проекту до определения его архитектуры, использует в качестве метрик количество строк кода или функциональные точки
Постархитектурная модель	Наиболее детализированная модель, используется после разработки архитектуры проекта и позволяет получить самые точные оценки, применяет в качестве метрик количество строк кода или функциональные пункты

В рамках этой модели оценки трудоемкости проекта и времени, требующегося на его выполнение, определяются тремя разными способами на разных этапах проекта:

1. На самых ранних этапах, когда примерно известны только общие требования, а проектирование еще не начиналось, используется модель состава приложения (Application Composition Model). В ее рамках трудоемкость проекта оценивается в человеко-месяцах по формуле:

$$PM = A \times Size \tag{20}$$

Size представляет собой оценку размера в терминах экранов, форм, отчетов, компонентов и модулей будущей системы. Коэффициент A учитывает возможное переиспользование части компонентов и производительность разработки, зависящую от опытности команды и используемых инструментов и оцениваемую числом от 4 до 50.

$$A = {(100 - (\%\ \text{переиспользования}))} \big/ {\text{Производительность}} \tag{21}$$

2. На следующих этапах, когда требования уже в основном известны и начинается разработка архитектуры ПО, используется модель этапа предварительного проектирования (Early Design Model) и следующие формулы.

Для трудоемкости (в человеко-месяцах):

$$PM = A \times Size^B \times \prod_i EM_i \tag{22}$$

Коэффициент A считается равным 2,45.

Size — оценка размера ПО в тысячах строк кода.

B — фактор процесса разработки, который вычисляется по формуле:

$$B = 0{,}91 + 0{,}01 \times \sum_i SF_i \tag{23}$$

где факторы (Scale Factors) SF_i принимают значения от 0 до 5 (таблица 6.12):

SF_1 — предсказуемость проекта для данной организации, от полностью знакомого (0) до совсем непредсказуемого (5);

SF_2 — гибкость процесса разработки, от полностью определяемого командой при выполнении общих целей проекта (0) до полностью фиксированного и строгого (5);

SF_3 — степень удаления рисков, от полной (0) до небольшой (5), оставляющей около 80% рисков;

SF_4 — сплоченность команды проекта, от безукоризненного взаимодействия (0) до больших трудностей при взаимодействии (5);

SF_5 — зрелость процессов в организации, от 0 до 5 в виде взвешенного количества положительных ответов на вопросы о поддержке ключевых областей процесса в модели CMM.

Таблица 6.13 - Scale Factors

Параметр	**Очень низкое (0)**	**Низкое (1)**	**Номина-льное (2)**	**Высокое (3)**	**Очень высокое (4)**	**Чрезвы-чайно высокое (5)**
PREC – Прецеден-тность	Полное отсутствие преце-дентов	Почти полное отсутствие прецеде-нтов	Наличие некоторого количества прецеде-нтов	Общее знаком-ство	Широкое знако-мство	Исчерпы-вающее знако-мство
FLEX – Гибкость разра-ботки	Строгая	Случайные послабле-ния	Некоторые послабле-ния	Общее соответ-тствие	Некоторое соответ-тствие	Общие цели
RESL – Разреше-ние рисков в архите-ктуре	Малое 20%	Некоторое 40%	Частое 60%	В целом 75%	Почти полное 90%	Полное 100%

Продолжение таблицы 6.13

TEAM – Сплоченность команды	Сильно затрудненное взаимодействие	Несколько затрудненное взаимодействие	Некоторая согласованность	Повышенная согласованность	Высокая согласованность	Взаимодействие как единого целого
PMAT – Зрелость процесса	Уровень 1	Уровень 2	Уровень 2+	Уровень 3	Уровень 4	Уровень 5

EM (Effort Multipliers) — произведение семи коэффициентов затрат, каждый из которых лежит в интервале от 1 до 6:

- возможности персонала;
- надежность и сложность продукта;
- требуемый уровень повторного использования;
- сложность платформы;
- опытность персонала;
- использование инструментов;
- плотность графика проекта.

Для времени (в месяцах):

$$TDEV = T \times PM^{(0,28+0,2\times(B-0,91))} \quad (24)$$

Коэффициент T равен 3,67. PM (Person Months) обозначает оценку трудоемкости.

3. После того, как разработана архитектура ПО, оценки должны выполняться с использованием постархитектурной модели (Post-Architecture Model).

Формула для трудоемкости (в человеко-месяцах):

$$PM = A \times (K_{req} \times Size)^{B} \times \prod_{i} EM_{i} \quad (25)$$

Для времени — та же формула, что и в предыдущей модели (в месяцах):

$$TDEV = T \times PM^{(0,28+0,2\times(B-0,91))} \qquad (26)$$

Коэффициент K_{req} вычисляется как:

$$K_{req} = 1 + (\text{\% выброшенного кода из - за изменения требований})/100 \qquad (27)$$

$$Size = (\text{новый код}) + (\text{переиспользуемый код}) \times \frac{100 - AT}{100} \times \frac{(AA + 0{,}4DM + 0{,}3CM + 0{,}3IM + SU)}{100}$$

(29)

где AT — процент автоматически генерируемого кода;

AA — фактор трудоемкости перевода компонентов в повторно используемые, от 0 до 8;

DM — процент модифицируемых для переиспользования проектных моделей;

CM — процент модифицируемого для переиспользования кода;

IM — процент затрат на интеграцию и тестирование повторно используемых компонентов;

SU — фактор понятности переиспользуемого кода, от 10 для простого, хорошо структурированного, до 50 для сложного и непонятного; равен 0, если CM = DM = 0.

EM_i – затратные коэффициенты (в Постархитектурной модели их 17):

Таблица 6.14 - Рейтинги и степень влияния регулирующих факторов COCOMO II

Фактор	Описание	Рейтинги						Влияние
		Очень низкие	Низкие	Номинальные	Высокие	Очень высокие	Чрезвычайно высокие	
Продукт								
RELY – Требуемая надежность программного обеспечения	Учитывает меру выполнения программой задуманного действия в течение определенного времени	0,82	0,92	1,00	1,10	1,26		1,54
DATA – Размер базы данных	Учитывает влияние объёма тестовых данных на разработку продукта. Уровень этого параметра рассчитывается как соотношение байт в тестируемой базе данных к SLOC в программе		0,90	1,00	1,14	1,28		1,42

Продолжение таблицы 6.14

RUSE – Разработка для повторного использования	Учитывает трудозатраты, требуемые дополнительно для написания компонентов, предназначенных для повторного использования в данном или последующих проектах. Использует следующие оценочные уровни: “в проекте”, “в программе”, “в линейке продуктов”, “в различных линейках продуктов”. Значение параметра накладывает ограничения на следующие параметры: RELY и DOCU		0,95	1,00	1,07	1,15	1,24	1,31
DOCU – Объем необходимой документации	Учитывает степень соответствия документации проекта его жизненному циклу	0,81	0,91	1,00	1,11	1,23		1,52
CPLX – Сложность продукта	Включает пять типов операций: управления, счетные, устройство-зависимые, управления данными, управления пользовательским интерфейсом. Уровень сложности это субъективное средне-взвешенное значение уровней типов операций	0,73	0,87	1,00	1,17	1,34	1,74	2,38

Продолжение таблицы 6.14

Платформа								
TIME – Ограничения по быстродействию	Учитывает временные ресурсы, используемые ПО, при выполнении поставленной задачи			1,00	1,11	1,29	1,63	1,63
STOR – Ограничения по объему хранимых данных	Учитывает процент использования хранилищ данных			1,00	1,05	1,17	1,46	1,46
PVOL – Неустойчивость платформы	Учитывает срок жизни платформы (комплекс аппаратного и программного обеспечения, который требуется для функционирования разрабатываемого ПО)		0,87	1,00	1,15	1,30		1,49
Персонал								
ACAP – Квалификация аналитиков по требованиям	Учитывает анализ, способность проектировать, эффективность и коммуникативные способности группы специалистов, которые разрабатывают требования и спецификации проекта. Параметр не должен оценивать уровень квалификации отдельно взятого специалиста	1,42	1,19	1,00	0,85	0,71		2,00
APEX – Опыт работы в прикладной области	Учитывает опыт коллектива при работе над приложениями определенного типа	1,22	1,10	1,00	0,80	0,81		1,51

Продолжение таблицы 6.14

PCAP - Квалификация программистов (общая)	Учитывает уровень программистов в коллективе. При выборе значения для этого параметра следует особо обратить внимание на коммуникативные и профессиональные способности программистов и на командную работу в целом	1,34	1,15	1,00	0,88	0,76		1,76
PEXP – Опыт разработки для платформы	Учитывает умение использовать особенности платформ, такие как графический интерфейс, базы данных, сетевой интерфейс, распределенные системы	1,19	1,09	1,00	0,91	0,85		1,40
LTEX – Навыки владения языками и инструментарием	Учитывает опыт программистов (языки, среды и инструменты)	1,20	1,09	1,00	0,91	0,84		1,43
PCON – Постоянство персонала	Учитывает текучесть кадров в коллективе	1,29	1,12	1,00	0,90	0,81		1,59
Проект								
TOOL – Использование программных инструментов	Учитывает уровень использования инструментов разработки	1,17	1,09	1,00	0,90	0,78		1,50

Продолжение таблицы 6.14

SITE – Распределенная разработка	Учитывает территориальную удаленность (от офиса до международных офисов) членов команды разработчиков и используемые ими средства коммуникации (от телефона до видео конференц-связи)	1,22	1,09	1,00	0,93	0,86	0,78	1,56
SCED – Сжатие графика проекта	Учитывает влияние временных ограничений, накладываемых на проект и на значение трудозатрат	1,43	1,14	1,00	1,00	1,00		1,43

6.8 Модель Путнэма (SLIM)

Вернемся к линейной формуле определения трудозатрат – как уже говорилось, она была признана несостоятельной, но если ранее сомнению подвергался только способ исчисления размера ПО, то теперь стоит задуматься о ее линейном характере.

О том, что трудоемкость и, соответственно, стоимость программного проекта нелинейно зависит от объема работ, было известно еще в 1970-х годах, когда появились первые научные публикации, подкрепленные результатами серьезных исследований.

Вероятно, первой нелинейной моделью, использующей эмпирические данные и нашедшей практическое применение при оценке стоимости ПО, стала SLIM (Software Life-cycle Model), предложенная в 1978 г. Лоуренсом Путнэмом.

SLIM была создана на базе реальных данных, собранных в Министерстве обороны США, и ориентирована в первую очередь на крупные проекты. Несмотря на возможность калибровки модели на

основе хронологической информации, что несколько повышает качество результатов, она не приобрела широкой популярности, хотя существуют организации, успешно использующие ее в проектном менеджменте и сегодня.

Созданная для проектов, объемом больше 70 000 строк кода, модель основывается на утверждении, что затраты на разработку ПО распределяются согласно кривым Нордена-Рэйли, которые являются графиками функции, представляющей распределение рабочей силы по времени. Общий вид подобной функции:

$$v = v_0 \cdot \left(1 - e^{-t^2 / 2t_p^2}\right), \qquad (28)$$

где v – полученное значение; t – время; v_0 и t_p – параметры, определяющие функцию. Для большого значения t, кривая стремится к параметру v_0, который называется cost scale factor parameter. Функция возрастает наиболее быстро при $t = t_p$. Основной причиной такого поведения модели являлось то, что изначально исследования Нордена базировались не на теоретической основе, а на наблюдениях за проектами, причем, в основном за проектами не связанными с ПО (машиностроение, строительство). Поэтому нет научного подтверждения тому, что программные проекты требуют такого же распределения рабочей силы, наоборот, зачастую количество человеко-часов, требуемых проектом, может резко измениться, сделав оценку непригодной к использованию. После ряда эмпирических наблюдений, Путнэм выразил рабочее уравнение модели в форме:

$$Size = C \cdot E^{1/3} \cdot t^{4/3}, \qquad (29)$$

где Size – размер кода в LOC, C – технологический фактор; E – общая стоимость проекта в человеко-годах; t – ожидаемое время реализации

проекта. Технологический фактор включает в себя характеристику проекта в следующих аспектах: методы управления и понимание процесса, качество используемых методов инженерии ПО, уровень используемых языков программирования, уровень развития среды, навыки и опыт команды разработчиков, сложность приложения.

Уравнение для E, выглядит как:

$$E = D_0 \times t_d^3, \tag{30}$$

где D_0 – коэффициент, выражающий количество необходимой работы (значения от 8 до 12, означает ПО полностью новое, с большим количеством связей; значения до 27 – требуется переработка существующего кода). Связывая два уравнения, получаем:

$$E = \left(D_0^{4/7} \quad C^{9/7} \right) \quad S^{9/7} \text{ и } t_d = \left(D_0^{-1/7} \cdot E^{-3/7} \right) \cdot S^{-3/7}. \tag{31}$$

В 1991 году Путнэмом была представлена альтернативная реализация модели, выполненная по заказу Quantitative Software Management (QSM) Inc. и примененная в комплексе SLIM Estimate для оценки стоимости ПО. Полное уравнение в этой реализации выглядит как:

$$E = 12^5 \quad B \quad \left(SLOC/P\right)^3 \quad \left(1/Schedule^4\right). \tag{32}$$

Если на общее время реализации проекта ограничения не накладываются, то возможно использование упрощенного уравнения:

$$E = 56.4 \quad B \quad \left(SLOC/P\right)^{\frac{9}{7}}. \tag{33}$$

Здесь *B* – фактор специальных навыков; *P* – табличный фактор продуктивности, зависящий от среды применения разрабатываемого приложения; Schedule – время разработки по графику (в месяцах). Уравнение может быть использовано, если предполагаемые затраты больше 20 человеко-месяцев.

6.9 Программные средства для оценки стоимости ПО

Оценка трудозатрат и стоимости по описанным выше моделям является достаточно сложной для ручного подсчета. Для автоматизации процесса оценки различными компаниями разработан широкий спектр оценочных программ, разброс цен на которые колеблется от бесплатного распространения до $20000 за готовую лицензию на одно рабочее место. Наиболее популярными программами являются:

- Angel (Analogy Software Tool) – программа с возможностью оценки будущих проектов по аналогии с прошлыми проектами;
- Construx Estimate – бесплатная программа, основанная на моделях Путнэма и COCOMO II, разработанная компанией Construx Software Builders (рис. 6.3);

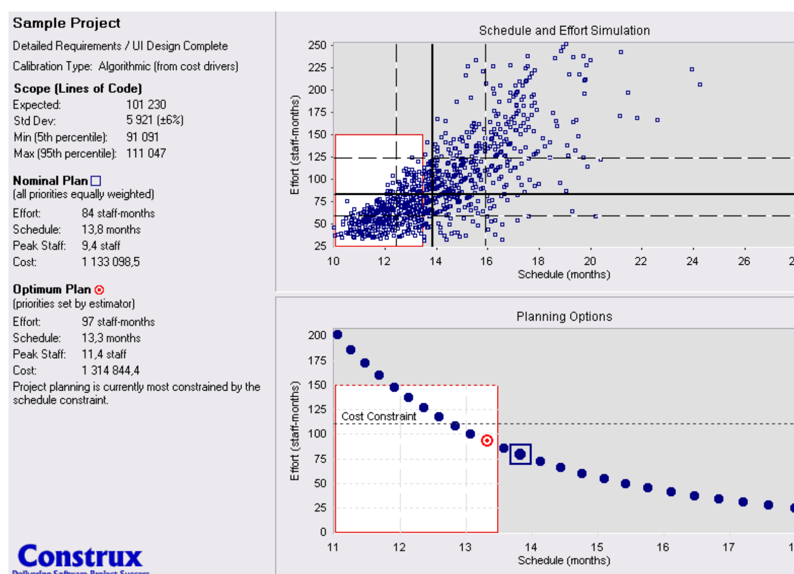

Рис. 6.3 - Окно Construx Estimate

- COCOMO II – программные реализации модели COCOMO II, официальные версии находятся на сайте Южнокалифорнийского университета (Http://sunset.usc.edu) и распространяются бесплатно (рис. 6.4);

COCOMO II with Heuristic Risk Assessment

Model: Post-architecture
Calibration: COCOMOII.2000
Current rule base implementation

Size

	SLOC	% Design Modified	% Code Modified	% Integration Required	Assessment and Assimilation (0% - 8%)	Software Understanding (0% - 50%)	Unfamiliarity (0-1)
New							
Reused		0	0				
Modified							

Rate each cost driver below from Very Low (VL) to Extra High (EH). For **HELP** on each cost driver, select its name.

	Very Low (VL)	Low (L)	Nominal (N)	High (H)	Very High (VH)	Extra High (EH)
Scale Drivers						
Precedentedness	VL	L	N	H	VH	XH
Development Flexibility	VL	L	N	H	VH	XH
Architecture/Risk Resolution	VL	L	N	H	VH	XH
Team Cohesion	VL	L	N	H	VH	XH
Process Maturity	VL	L	N	H	VH	XH
Product Attributes						
Required Reliability	VL	L	N	H	VH	
Database Size		L	N	H	VH	
Product Complexity	VL	L	N	H	VH	EH
Required Reuse		L	N	H	VH	EH
Documentation	VL	L	N	H	VH	
Platform Attributes						
Execution Time Constraint			N	H	VH	EH
Main Storage Constraint			N	H	VH	EH
Platform Volatility		L	N	H	VH	
Personnel Attributes						
Analyst Capability	VL	L	N	H	VH	
Programmer Capability	VL	L	N	H	VH	
Personnel Continuity	VL	L	N	H	VH	
Applications Experience	VL	L	N	H	VH	
Platform Experience	VL	L	N	H	VH	
Language and Toolset Experience	VL	L	N	H	VH	
Project Attributes						
Use of Software Tools	VL	L	N	H	VH	
Multisite Development	VL	L	N	H	VH	XH
Required Development Schedule	VL	L	N	H	VH	

Submit | Сброс

Рис. 6.4 - Программная реализация модели COCOMO II

- Costar – недорогая полноценная реализация COCOMO II, предлагаемая компанией Softstar Systems (рис. 6.5);

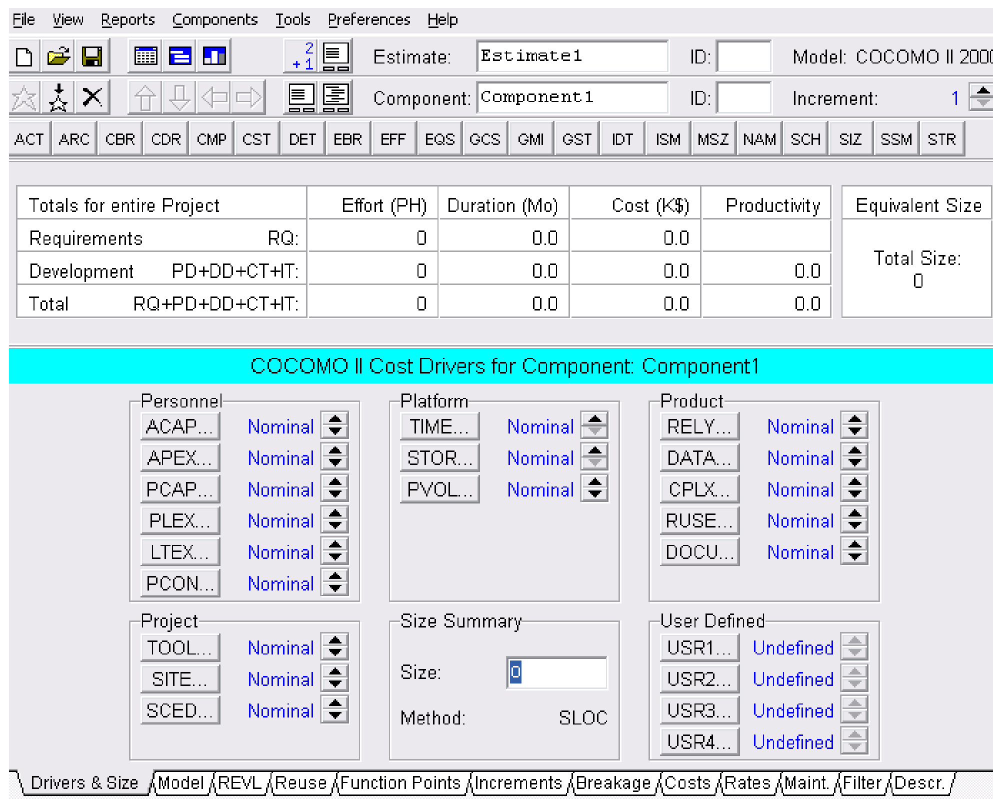

Рис. 6.5 - Программная реализация модели COCOMO II

- KnowledgePLAN – программа, разработанная компанией Software Productivity Research и отличающаяся высокой интеграцией с Microsoft Project;
- Price-S – программа представляет собой пакет программных продуктов, предназначенных для оценки проектов;
- SEER – как и Price-S состоит из нескольких взаимосвязанных продуктов: SEER-SEM – оценка, планирование и управление, SEER-SSM – углубленная оценка размеров программных проектов, SEER-AccuScope – простая оценка размеров;
- SLIM-Estimate и Estimate Express – семейство программных продуктов от Quantative Software Management, являющихся мощными и полнофункциональными оценочными программами, базирующиеся на модели оценки Путнэма.

Вопросы к главе 6

1. В чем заключается метод оценки по аналогии?
2. В чем заключается метод параметрических оценок?
3. Как осуществляется декомпозиция проекта с применением структуры трудозатрат?
4. В чем заключается метод экспертных оценок?
5. Какие методы получения групповых экспертных оценок Вы знаете?
6. Перечислите основные шаги дельфийского метода.
7. Опишите линейный подход к оценке стоимости ПО.
8. Как определяется цена одной единицы трудозатрат?
9. Что такое функциональные пункты?
10. Какие методы оценки стоимости ПО, основанные на функциональных пунктах Вы знаете?
11. Какие методы оценки стоимости ПО, основаны на использовании эмпирических данных Вы знаете?
12. Перечислите режимы модели COCOMO.
13. Опишите метод IFPUG.
14. Перечислите факторы влияния в методе IFPUG.
15. Как можно использовать элементы пользовательского интерфейса в методе функциональных пунктов?
16. Как перевести функциональные пункты в человеко-месяцы?
17. На каких этапах проекта используется модель COCOMO II?
18. Опишите модель Путнема.
19. Какие программные средства, применяемые для оценки стоимости ПО Вы знаете?

ЗАКЛЮЧЕНИЕ

В учебном пособии были рассмотрены теоретические вопросы, касающиеся процесса разработки программного обеспечения и связанные с ним проблемы.

В работе проведена попытка описать и провести анализ современных методов и подходов к организации жизненного цикла разработки программного обеспечения, включая описание современных стандартов, методологий разработки, методов стоимостной оценки программного обеспечения, подходов к процессу тестирования программного обеспечения.

Отдельное внимание в пособии было уделено основным историческим этапам, давшим толчок развитию теории управления процессом разработки программного обеспечения.

Материал, приведенный в пособии даёт понять, что процесс разработки программного обеспечения это не только написание программного кода, но и технологический процесс, состоящий из определённого набора видов деятельности, каждый из которых важен по-своему для создания качественного программного обеспечения.

Пренебрежение одним из этих аспектов, таких как принципы методологии разработки, тестирование, нарушение стандартов в области качества, стоимостная оценка, неминуемо приведет к созданию некачественного программного обеспечения либо к краху проекта по его созданию.

В процессе разработки учебного пособия авторами рассмотрены теоретические вопросы по каждому из этапов разработки программного обеспечения, позволяющие оценить уровень знаний студентов по вопросам, связанным с жизненным циклом разработки ПО, и применения этих знаний на практике.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. ГОСТ Р ISO/МЭК 12207. Процессы жизненного цикла программных средств : утвержден и введен в действие Приказом Федерального агентства по техническому регулированию и метрологии от 30 ноября 2010 г. № 631-ст.

2. Александров, А. А. Сертифицированный тестировщик [Текст] : Программа обучения базового уровня / А. А. Александров, А.В. Конушин. – М.: RSTQB, 2011. – 101 с.

3. Благодатских, В. А. Стандартизация разработки программных средств [Текст] : учебное пособие / В. А. Благодатских, В. А. Волнин, К. Ф. Поскака-лов; под ред. О. С. Разумова. - М.: Финансы и статистика, 2005. - 288 с. ISBN 5-279-02657-3

4. Вендров, А. М. Проектирование программного обеспечения экономиче-ских информационных систем [Текст] : учебник. — М.: Финансы и стати-стика, 2000. — 352 с.

5. Гагарина, Л. Г. Технология разработки программного обеспечения [Текст]: учебное пособие / Л. Г. Гагарина, Е. В. Кокорева, Б. Д. Сидорова-Виснадул ; под ред. Л. Г. Гагариной. — Москва : ИД «ФОРУМ» : ИНФРА-М, 2018. — 400 с. — (Высшее образование: Бакалавриат). — ISBN 978-5-16-104071-3.

6. Гласе, Р. Руководство по надежному программированию [Текст] : Р. Гласе; пер. с англ. Ю. П. Кондранина, В. М. Рабиновича ; под ред. В. М. Рабиновича; предисл. В.В. Липаева. — М.: Финансы и статистика, 1982. — 256 с.

7. Липаев, В. В. Надежность программных средств [Текст] : В. В. Липаев. — М.: СИНТЕГ, 1998. — 232 с. — (Серия «Информатизация России на пороге XXI века»).

8. Майерс, Г. Искусство тестирования программ [Текст] : Г. Майерс ; пер. с англ. ; под ред. Б. А. Позина. — М.: Финансы и статистика, 1982. — 176 с.

9. Майерс, Г. Надежность программного обеспечения [Текст] : Г. Майерс ; пер. с англ. Ю. Ю. Галимова ; под ред. В. Ш. Кауфмана. — М.: Мир, 1980. — 361 с.

10. Макконнелл, С. Сколько стоит программный проект [Текст] : С. Макконнелл.– М.: «Русская редакция», СПб.: Питер, 2007. – 297 с.

11. Михайловский, Н. Э. Сравнение методов оценки стоимости проектов по разработке информационных систем [Электронный ресурс] / Н. Э. Михайловский. – Режим доступа: www.ntrlab.ru

12. Першиков, В. И. Толковый словарь по информатике [Текст] : В. И. Першиков, В. М. Савинков. — 2-е изд., доп. — М.: Финансы и статистика, 1995. — 544 с.

13. Саймон, А. Р. Стратегические технологии баз данных: менеджмент на 2000 год [Текст] : А. Р. Саймон ; пер. с англ. ; под ред. и с предисл. М. Р. Когаловского. — М.: Финансы и статистика, 1999. — 479 с.

14. Терехов, А. Н. Технология программирования [Текст] : учеб. пособие по специальности ≪Математ. обеспечение и администрирование информ. систем≫ – 010503 / А. Н. Терехов. – М.: Интернет-Ун-т Информ. Технологий, 2006. – 152 с. - (Серия ≪Информационные технологии: от первого лица≫ / Интернет-Ун-т Информ. Технологий). – ISBN 5-9556-0048-5.

15. Технология программирования : учебное пособие / Ю.Ю. Громов [и др.]. – Тамбов : Изд-во ФГБОУ ВПО «ТГТУ», 2013. – 172 с. – 100 экз. ISBN 978-5-8265-1207-4

16. Boehm B.W. Software engineering economics / B. W. Boehm. – Prentice-Hall, 1981. – P. 320.

17. Parkinson S. N. Parkinson's Law and Other Studies in Administration / S. N. Parkinson. – Houghton-Miffin, 1957. – P. 148.
18. COCOMO II Model Definition Manual Fundamentals of Function Point Analysis [Электронный ресурс]. — Режим доступа : www.softwaremetrics.com/fpafund.htm, свободный.
19. Гибкая разработка, кратко о методологии Agile [Электронный ресурс] / Хабр. — Режим доступа : https://habr.com/ru/company/it-guild/blog/341924/, свободный.
20. Ещё раз про семь основных методологий разработки [Электронный ресурс] / Хабр. — Режим доступа : https://habr.com/ru/company/edison/blog/269789/, свободный.
21. Жизненный цикл программного обеспечения [Электронный ресурс] / Википедия. – Режим доступа : https://ru.wikipedia.org/wiki/Жизненный цикл программного обеспечения, свободный.
22. Жизненный цикл ПО. Каскадная модель (Waterfall) [Электронный ресурс] / XB Software Блог. — Режим доступа : https://xbsoftware.ru/blog/zhiznennyj-tsykl-po-kaskadnaya-model-waterfall/, свободный.
23. Жизненный цикл разработки ПО (SDLC). Спиральная модель [Электронный ресурс] / XB Software Блог. — Режим доступа : https://xbsoftware.ru/blog/zhiznennyj-tsykl-razrabotki-spiral/, свободный.
24. Итеративная модель [Электронный ресурс]. — Режим доступа : https://qalight.com.ua/baza-znaniy/iterativnaya-model-iterative-model, свободный.
25. Семь методов управления проектами [Электронный ресурс]. — Режим доступа : https://www.pmservices.ru/project-management-news/top-7-metodov-upravleniya-proektami-agile-scrum-kanban-prince2-i-drugie/, свободный.

26. Спиральная модель ЖЦ [Электронный ресурс]. — Режим доступа : https://qalight.com.ua/baza-znaniy/spiralnaya-model-spiral-model/, свободный.
27. V-образная модель [Электронный ресурс]. — Режим доступа : https://studopedia.net/1_42154_V-obraznaya-model-preimushchestva-nedostatki-oblast primeneniya.html, свободный.
28. V-образная модель [Электронный ресурс]. — Режим доступа : https://qalight.com.ua/baza-znaniy/v-model-v-model/, свободный.
29. 12 методологий разработки ПО [Электронный ресурс]. — Режим доступа: https://geekbrains.ru/posts/methodologies, свободный.

Учебное электронное издание

ГРАДУСОВ Денис Александрович
ШУТОВ Антон Владимирович

ТЕОРЕТИЧЕСКИЕ ВОПРОСЫ РАЗРАБОТКИ
ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Издается в авторской редакции

Системные требования: Intel от 1,3 ГГц ; Windows XP/7/8/10; Adobe Reader; дисковод DVD-ROM.

Тираж 35 экз.

Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых
Изд-во ВлГУ
rio.vlgu@yandex.ru

Институт информационных технологий и радиоэлектроники
Кафедра вычислительной техники и систем управления
Lantsov@VLSU.ru