КЛАССИКА COMPUTER SCIENCE

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ЭРИК ДЖ. БРАУДЕ

WILEY

ПИТЕР®

СЕРИЯ

ПИТЕР®

SOFTWARE ENGINEERING: An Object-Oriented Perspective

Eric J. Braude

Wiley Computer Publishing
John Wiley & Sons, Inc.
New York ▪ Chichester ▪ Weinheim ▪ Brisbane ▪ Singapore ▪ Toronto

КЛАССИКА COMPUTER SCIENCE

ЭРИК ДЖ. БРАУДЕ

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Новосибирск · Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2004

ББК 32.973.2-018
УДК 681.3.06
Б87

Брауде Э.

Б87 Технология разработки программного обеспечения. — СПб.: Питер, 2004. — 655 с.: ил.

ISBN 5-94723-663-X

Книга является классическим руководством по технологии разработки программного обеспечения. В ней рассматриваются основные этапы процесса разработки программного продукта, включая его описание, проектирование, собственно разработку, интеграцию и тестирование частей, тестирование программы в целом и сопровождение. Каждый из этапов подробно иллюстрируется на конкретных примерах. Отдельные главы посвящены определению типа процесса разработки и управлению проектом. Материал сопровождается упражнениями для закрепления теоретических знаний.

Предназначена для студентов и аспирантов, специализирующихся в области программного обеспечения, а также для профессиональных разработчиков.

ББК 32.973.2-018
УДК 681.3.06

ISBN 0471322083 (англ.)
ISBN 5-94723-663-X

Краткое содержание

Содержание

Джуди, Майклу, Мириам и Рэйчел

«Я бросил мечты под ноги тебе...»
У. Б. Йетс

Предисловие

Данная книга — это не просто книга о разработке программного обеспечения; лучше сказать, что это учебник по исследованию методов разработки программного обеспечения. Замысел этой книги возник давно, но не мог найти своего полного осуществления, поскольку не было достаточно широко принятого, общего среди специалистов технического подхода. Несмотря на то, что многие современные проекты используют объектно-ориентированные языки, объектно-ориентированному подходу в сущности уделялось лишь немного места в учебниках по разработке программного обеспечения.

В 90-х годах Сообщество объектно-ориентированного анализа и разработки (Object-Oriented Analysis and Design community) выработало подход к разработке прикладных программ и дало ему соответствующее название — *Унифицированный язык моделирования* (UML). После того как этот язык и подход получили широкое признание, у нас появились все основания считать новое тысячелетие временем для обучения и научения проектированию программного обеспечения, а не только временем для разговоров о разработке программного обеспечения. Таким образом, несмотря на то, что в данной книге есть аспекты разработки программного обеспечения, которые не являются объетно-ориентированными, по своей сути книга предоставляет читателю учебный материал именно по объектно-ориентированному подходу и дает знания по применению каркасов, примеров и образцов проектирования. В ней также исследуются вопросы отношения объектно-ориентированного программирования к анализу и тестированию требований. Благодаря этому преподаватели потратят меньше времени на освещение множества различных подходов и смогут уделить больше внимания углубленному изучению предмета и проведению практической работы.

Любая книга, посвященная разработке программного обеспечения, обязательно включает в себя изучение примеров. Вдобавок, поскольку разработка программного обеспечения главным образом связана с комплексным подходом и сложностью, в учебнике по разработке достаточно большой, отнюдь не символический, объем должен быть отведен глубокому изучению примеров. И наконец, изучение примеров должно быть интересно для учеников — им нужно четко представлять себе постройку на этом фундаменте, уметь легко на нем конструировать. Именно с этой целью принципы разработки программного обеспечения демонстрируются в данной книге на примере создания ролевой видеоигры. Видеоигры обладают широкими возможностями для демонстрации каркасов, образцов проектирования, состояний, параллельных процессов и нетривиальных графических

пользовательских интерфейсов. Изучение примеров дополняют и завершают примеры научных и коммерческих разработок.

Рабочий программный продукт для этой книги написан командой программистов-разработчиков, а не одним человеком. Книга будет также весьма полезна для студенческих команд. Автора побудила к данной работе первооткрывательская работа У. Хэмфри, посвященная Унифицированному процессу разработки программного обеспечения (USDP) [Hu], и работа о Процессе командной разработки программного обеспечения (TSP) [Hu7].

Круг читателей

Эта книга предназначена для студентов старших курсов и аспирантов первого года обучения, специализирующихся в данной области. Поскольку цель книги — наработать навык и достичь умений в разработке программного обеспечения, то книга также станет полезной и для профессиональных разработчиков программного обеспечения, которые хотят расширить свои познания, усовершенствовать навыки и способности. Предполагается, что читателю уже знакомо программирование с использованием классов и объектов, в основном на языке Java.

Структура книги

Каждая из глав 1, 2, 3, 4, 5 и 6 разделена на две части. Такое деление способствует быстрому ознакомлению читателей и преподавателей, стремящихся к скорому прогрессу, с основополагающими знаниями по анализу требований и разработке. С этой целью читателю предлагается для начала прочесть первые части этих глав книги, а потом вернуться ко вторым их частям.

- Во введении дается краткий обзор разработки программного обеспечения, а также советы преподавателям для обучения групп студентов.
- В главе 1 дается подробное описание процесса разработки программного обеспечения. Раздел 1.6 является основной и фундаментально важной частью данной главы.
- В главе 2 объясняется, как построены программные проекты. Специалисты в области техники часто пропускают такие места в книге. Однако если они будут работать в группе и дойдут до сути вопросов построения и управления, то данную главу они прочтут с пользой для дела. В частности, в разделе 2.1, во введении и в разделе 2.4 (посвященном риску) обсуждаются существенно важные для разработчика программного обеспечения вопросы.

Главы с 3 по 10 идут в таком же логическом порядке, в каком создается программный продукт на каждой итерации.

- В главах 3 и 4 освещается анализ требований. На этом этапе требуется выяснить, что необходимо создать.

- В главах 5 и 6 объясняется, как разрабатывать продукт и какие при этом используются средства разработки.
- В главе 7 объясняется программирование в контексте разработки программного обеспечения.
- Главы 8 и 9 посвящены процессу тестирования.
- В главе 10 обсуждаются действия, необходимые, когда программный продукт уже выпущен.
- В конце книги приведена библиография, список сокращений и словарь терминов.

Как пользоваться книгой

Хотя главы этой книги следуют в том же логическом порядке, что и этапы процесса разработки, он тем не менее только отчасти отражает последовательность создания приложений. Последовательность *анализ требований — проектирование — программирование — тестирование* обычно повторяется по крайней мере один раз. В главах 1 и 2 обсуждается, как можно организовать это построение.

Есть несколько основных способов использования этой книги, и каждый из них имеет свою мотивировку и свои приоритеты. Ниже приводится перечень этих способов.

- **Линейный способ. Читать главы в порядке их следования: введение, главы 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.** Автор сам использовал материал этой книги в своей преподавательской работе. С этой целью было взято предисловие и введение, а далее — главы с 1 по 10 последовательно. В главах 1 и 2 делался общий обзор, поскольку перед студенческими командами первым делом стоит задача пройти составляющие элементы последовательности *анализ требований — проектирование — программирование — тестирование* с использованием *тривиального* набора требований. Это приучает студентов к понятию *процесса*, побуждает их к взаимодействию внутри команды, благодаря чему перед ними в контексте групповой работы возникают актуальные задачи. Но после главы 2 данный метод во многом теряет свою эффективность.
- **Способ «двойного зачета». Читать главы в следующем порядке: введение, первые части глав 1, 2, 3, 4, 5, 6, затем вторые части глав 1, 2, 3, 4, 5, 6, далее главы 7, 8, 9, 10.** Первый «зачет» складывается из вводных частей глав, второй «зачет» происходит после изучения оставшихся частей глав до конца. Такая очередность имеет свое преимущество: по прочтении первых частей глав студенты в группах сумеют построить небольшой прототип, а затем на его основе перейти к процессу полномасштабной разработки. Автор советует сделать данный прототип по возможности сжатым и лаконичным. Основная задача прототипа состоит в том, чтобы студенты

работали в команде, и на конкретных примерах и в конкретных ситуациях могли бы проработать разные стороны процесса разработки. Выполнение подобной работы отнимает много времени у новичков.

- **Способ «лестница карьеры». Читать главы следует в таком порядке: введение, главы 1, 7, 8, 6, 5, 2, 3, 4, 9, 10.** После главы 1, в которой даны ознакомительные сведения и вводные понятия по разработке программного обеспечения, преподаватель может пройти по всем главам в соответствии с типичной карьерой разработчика программного обеспечения компании. Программист начинает карьеру в роли *простого программиста* (этому начальному этапу соответствуют главы 7 и 8 и в качестве основного примера такой разработки взят пример из главы 6). Программист в конечном итоге приобретает статус и ответственность *проектировщика* или *разработчика* (глава 6 с использованием архитектуры примеров главы 4). Как это обычно и происходит, *проектировщик* со временем становится *архитектором* (глава 5 с использованием требований из примера в главе 6). И завершающий этап карьеры, на который и рассчитана книга, — это *руководитель проекта* (главы 2, 3, 4, 9 и 10).

Веб-сайт этой книги

По адресу http://www.wiley.com/college/braude вы найдете следующие сведения из книги:

- цветные слайды всех рисунков и списков данной книги, выполненные в формате PowerPoint. Это позволит преподавателям вносить изменения и настраивать слайды, а также объединять их по своему выбору;
- ответы на общие упражнения (в режиме, защищенном паролем);
- исходный код на языке Java для изучения примеров по курсу данной книги;
- документация по примерам, которую можно использовать в качестве шаблонов.

С этим веб-сайтом связано много планов, и он все время обновляется. Именно поэтому по вопросам поиска наиболее современных сведений читателю рекомендуется обращаться к веб-сайту.

Упражнения

В каждой главе есть три типа упражнений. *Вопросы для проверки* снабжены короткими ответами или подсказками, которые даны в той же главе, что и вопросы. В *командных упражнениях* сформулированы конкретные задачи и оценочные критерии для команд, работающих над проектами семестра. Решения упражнений для третьего типа — *общих* — в книге не приводятся, однако преподаватели могут найти эти решения на веб-сайте этой книги.

Благодарности

Опыт, который я приобрел, будучи разработчиком программного обеспечения и менеджером, и моя нынешняя научная деятельность в качестве профессора и консультанта по вопросам программирования, показали мне, насколько широко распространено среди профессионалов стремление осуществлять разработку программного обеспечения *правильно* с самого начала, несмотря на неослабевающее давление бизнеса. Я благодарен своим коллегам и студентам за то, что они посвятили так много времени работе вместе со мной при определении основных требований к разработке программного обеспечения.

Я особенно благодарен Дику Боствику за помощь, которую он оказывал мне на каждом этапе написания книги. Я обязан Тому ван Курту за его широкую и самоотверженную поддержку. Я хочу поблагодарить своих студентов колледжа Метрополитен Бостонского университета. Спасибо вам за ваш вклад в работу. Рецензентам книги: ваши комментарии и рецензии сделали книгу во многих отношениях лучше. Рецензентами книги были: Генри А. Этлингер (Институт технологии, Рочестер), Майкл Годфри (Университет Ватерлоо); Дэвид А. Густавсон (Государственный университет Канзаса), Питер Хичкок (компания DalTech, Университет Дальхузи), Флойд Лекюро (Государственный университет Калифорнии, Сакраменто), Стивен П. Райс (Университет Брауна), Лори Верт (Университет Техаса, Остин). Мне хочется поблагодарить своих коллег и администрацию колледжа Метрополитен Бостонского университета за проявленный интерес, воодушевление и энтузиазм. Я очень благодарен Полу Крокету, Дженни Велтер и Биллу Зобристу из компании John Wiley & Sons, а также Яну Фишеру из Publication Services за скрупулезную работу вместе со мной над данным проектом.

И наконец, я хотел бы засвидетельствовать, хотя и не вполне соразмерно оказанной поддержке, глубокую благодарность своей жене Джуди, а также моему сыну Майклу за то, что они поддерживали меня и сочувствовали моей страстной увлеченности этой книгой.

Эрик Дж. Брауде
Университет Бостона
Метрополитен колледж
Бостон, Массачусетс; апрель 2000

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция). Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на веб-сайте издательства http://www.piter.com.

Введение

И начинанья, взнесшиеся мощно...
Шекспир. Гамлет

Введение рассказывает о том, что такое технология разработки программного продукта и как организована эта книга.

Создание крупных программных приложений является одной из важнейших инженерных задач нашего времени.

0.1. Контекст разработки программного продукта

Технология разработки программных продуктов — это по определению одна из областей инженерной науки, и поэтому она несет такую же социальную ответственность, как и другие области.

С начала развития компьютерных технологий работу по созданию программных продуктов относили к *разработке*, для которой требуются в основном навыки программирования, а не знания инженерной науки. Аккредитационный совет по инженерной науке и технологии (ABET — Accreditation Board for Engineering and Technology) определяет профессию инженера следующим образом [1].

«*Профессия*, в которой *математические* и *естественно-научные* знания, полученные исследованиями, опытом и практикой, *мудро применяются* для разработки путей экономного *использования природных ресурсов и сил на пользу человечеству*.»

Много труда было вложено в развитие инженерной науки еще до рождения первого программного продукта. Сейчас, в начале третьего тысячелетия, процесс разработки программного продукта начинает требовать от своих создателей такой же высоты научных знаний, как это необходимо в других областях инженерной науки — электронике, механике или строительстве. Темой этой книги является сущность процесса разработки программного продукта.

В чем сходство и различие между разработкой программного продукта и другими областями инженерной науки? Один общий для них элемент — это необходимость подробного описания того, что должно быть создано, так называемый *анализ требований*. С другой стороны, программные проекты особенно часто подвергаются изменениям, включая те, которые необходимо сделать, пока продукт находится еще на стадии разработки.

0.2. Этапы разработки программного обеспечения

В 80-е и 90-е годы в области разработки программного обеспечения (ПО) преобладали две тенденции. Одна — это быстрый рост приложений, в том числе создаваемых для Web. Другая — это расцвет инструментальных средств и парадигм (подходов к проектированию, таких как объектно-ориентированный).

Однако, несмотря на появление новых тенденций, основные этапы разработки программного обеспечения остались неизменными, как показано ниже.

- *Определение* процесса разработки ПО, который будет использоваться в дальнейшем, — глава 1.
- *Управление* проектом разработки — базовые сведения в главе 2, продолжение в следующих главах.
- *Описание* целевого программного продукта — главы 3 и 4.
- *Проектирование* продукта — главы 5 и 6.
- *Разработка* продукта, то есть его программирование — глава 7.
- *Тестирование* частей продукта — глава 8.
- *Интеграция* частей и тестирование продукта в целом — глава 9.
- *Сопровождение* продукта — глава 10.

Разработчики меняют последовательность проработки этих направлений и долю внимания, уделяемого каждому направлению, как показано в главе 2. В реальности разработка программного обеспечения обычно определяется требуемым набором функций или сроком сдачи проекта, что диктуется ситуацией на рынке. В результате только хорошо организованные группы инженеров, владеющие методами разработки программного обеспечения, способны правильно построить работу. В противном случае разработчиков обычно ожидает хаос и крах.

Система разработки программного обеспечения включает в себя *персонал*, *процесс*, *проект* и *продукт* (рис. 0.1). Использованные обозначения взяты из книги «Унифицированный процесс разработки программного обеспечения» (USDP — Unified Software Development Process) Якобсона, Буча и Рамбо [64]. USDP — один из процессов для разработки программного обеспечения — описан в этой книге. Таблица, изображенная в разделе «Процесс» на рис. 0.1, объясняется в главе 1. Диаграмма в разделе «Проект» показывает инженеров, занимающихся различной работой в соответствии с их обязанностями, а потом передающих результаты другим инженерам, продолжающим их работу. Раздел «Продукт» содержит гораздо больше, чем просто объектные модули и исходный код. Например, туда включаются также документация, результаты тестов и измерений продуктивности. В соответствии с USDP мы будем называть эти продукты *артефактами*. В этой книге описывается, что должен содержать полный набор артефактов.

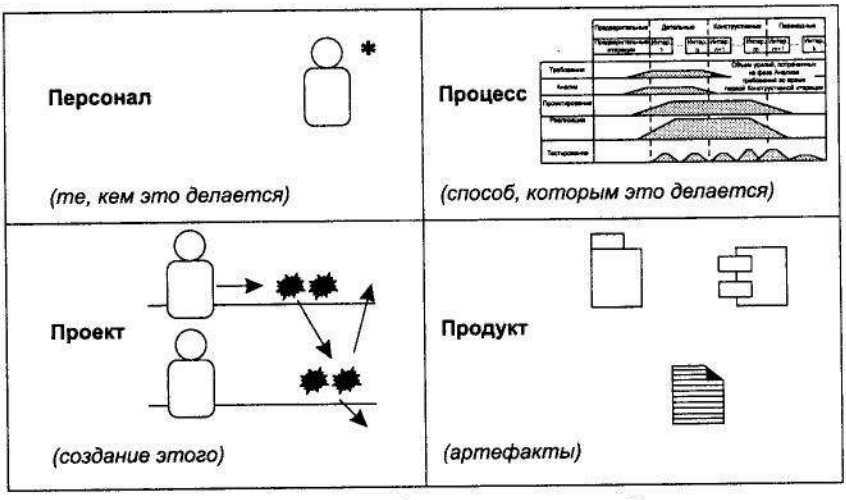


Рис. 0.1. Четыре «П» разработки программного обеспечения

0.3. Процесс

Содержание этого раздела представлено ниже.

- Процесс (главы 1 и 2).
- Последовательность разработки:
  - водопадная;
  - итеративная.
- Системы принципов разработки:
  - Индивидуальный процесс разработки (PSP);
  - Командный процесс разработки (TSP);
  - Модель зрелости возможностей (CMM).
- Стандарты:
  - Institute of Electrical and Electronic Engineering (IEEE) — Институт инженеров по электротехнике и радиоэлектронике;
  - International Standards Organization (ISO) — Международная организация по стандартизации.

Водопадный процесс (waterflow process) начинается с определения требований, предъявляемых приложению, далее переходит в фазу проектирования, затем реализации и, наконец, тестирования. Иногда в водопадный процесс включается фаза сопровождения, описанная в главе 10. В этой книге мы разделили *проектирование* на *архитектуру* (глава 5) и *детальное проектирование* (глава 6), а *тестирование* — на *модульное тестирование* (глава 8) и *системное тестирование* (глава 9). Разработка программного обеспечения редко протекает строго в «водопадной» последовательности. Процесс разработки продуктов для Web, например, имеет тенденцию менять направление, перемещаясь между спецификацией, проектирова-

нием, реализацией и тестированием. Также для разработки программного обеспечения мы часто используем *итеративные* процессы, в которых «водопад» повторяется несколько раз целиком или частично. Этот вопрос разбирается в главе 1. Представленный в строгой форме итеративный стиль имеет множество преимуществ.

Решения по поводу процесса разработки программного обеспечения часто принимаются на организационном уровне (уровне компании, отдела, группы разработчиков и т. д.), так что важно правильно оценивать потенциальные производительные возможности организации. В нескольких главах обсуждается такая оценка и приводится *модель зрелости возможностей* (CMM — Capability Maturity Model). Модель зрелости возможностей, созданная Уоттсом Хэмфри и Институтом технологий разработки программного обеспечения (SEI), описывается в главе 1.

У отдельных инженеров способности по разработке программного обеспечения могут быть развиты и оценены при помощи *индивидуального процесса разработки программного обеспечения* (PSP — Personal Software Process), созданного Хэмфри. Идеи CMM и PSP рассматриваются в нескольких главах этой книги. Третий уровень организации программного обеспечения — это *командный процесс разработки программного обеспечения* (TSP — Team Software Process) [55], описывающий процесс, следуя которому, команда разработчиков программного обеспечения выполняет свою работу. Мы уверены в том, что структурирование работы согласно системе принципов, такой как CMM, PSP и TSP, является основой профессиональной деятельности разработчика программного обеспечения в двадцать первом веке.

Хорошо продуманные стандарты документации облегчают создание полезных, надежных артефактов. В основном в этой книге используются стандарты IEEE, многие из которых одобрены Американским национальным институтом стандартов (ANSI — American National Standards Institute). Многие компании создают свои собственные стандарты. Хотя стандарты должны изменяться со временем, чтобы соответствовать развитию технологий, основа качественных стандартов остается неизменной в течение нескольких лет. Если команды разработчиков не используют стандарты или примеры, построенные на основе стандартов, то они теряют много времени в размышлениях над структурой (а не над сущностью) документа. Стандарты направляют процесс, обеспечивая инженеров, инструкторов и студентов «путеводной нитью» в их работе. На практике стандарты обычно изменяют и приспосабливают к конкретному проекту.

0.4. Проект

Проект — это совокупность действий, необходимая для создания артефакта. Проект включает контакт с заказчиком, написание документации, проектирование, написание кода и тестирование продукта. Избранные аспекты проекта представлены ниже.

- Объектно-ориентированный подход: очень полезная парадигма.
- Унифицированный язык моделирования (UML): нотация для проектирования.
- Унаследованные системы: отправная точка для улучшения или расширения существующей системы.

Объектно-ориентированный подход может быть очень полезен при разработке проекта. Он особенно эффективен при наличии постоянных изменений внутри проекта, так как помогает распределить сущности проектирования и исходный код по частям (классам и модулям), лучше отражающим проблему предметной области.

Унифицированный язык моделирования (UML — Unified Modeling Language, см. [90]) — промышленный стандарт для описания моделей — будет использоваться на протяжении всей этой книги. Обратите внимание, что UML — это не методология, а система обозначений.

В главе 5 исследуются способы разработки архитектуры приложения. Описанный подход заимствует идеи из области изучения образцов проектирования и из исследований по классификации архитектуры программного обеспечения. Глава 6 завершает обсуждение проектирования, демонстрируя, как могут быть указаны конкретные детали. Главы с 7 по 9 посвящены интеграции и тестированию приложений. В главе 10 обсуждаются сопровождение продукта, последняя и продолжающаяся стадии проекта.

Огромное количество разработок не нацелены на создание новых систем, а призваны улучшить или должны использовать уже существующие (*унаследованные*) системы. Даже представляющиеся абсолютно новыми приложения обычно должны сосуществовать с унаследованными системами. Однако трудно понять, как работать с унаследованными системами, не имея представлений о том, как в принципе должны создаваться системы. Поэтому эта книга ориентирована на новые, не унаследованные приложения. Читателям, желающим приобрести опыт использования унаследованных систем, будет полезно изучить законченный пример игры *Встреча* и на ее основе создать свою видеоигру.

0.5. Персонал

Взаимоотношения *людей*, занимающихся созданием программного обеспечения, существенным образом сказываются на успешности проекта. В этой книге мы не будем подробно обсуждать вопросы управления *персоналом*, но рассмотрим некоторые ситуации, жизненно важные для развития программного проекта. Команда разработчиков работает наилучшим образом, если каждый участник знает, что он должен делать, и имеет определенные обязанности. В книге представлено множество примеров, иллюстрирующих данный вопрос. *Командный процесс разработки программного обеспечения* (TSP) имеет непосредственное отношение к данному вопросу, и в главе 2 среди прочего мы представили полезную информацию об управлении командами разработчиков.

Другая сторона аспекта *персонала* — это заинтересованные в проекте лица: заказчики, пользователи и инвесторы. Хотя их влияние на проект может быть очень велико, рассмотрение работы с этими людьми не входит в задачи данной книги.

0.6. Продукт

Продукт, включая не только программное приложение, но и все составляющие его артефакты, представлен в табл. 0.1.

Таблица 0.1. Продукт: приложение и связанные с ним артефакты

Составляющие продукта	Артефакты
Требования (главы 3 и 4): опишите, что должен представлять собой продукт	Спецификация требований к программному обеспечению (SRS)
Программная архитектура (глава 5): используйте классификацию Гарлана и Шоу	Проектная модель
Детальное проектирование (глава 6): используйте образцы проектирования	
Реализация (глава 7): делайте акцент на использовании стандартов, применяйте избранные формальные методы	Исходный и объектный код
Артефакты тестирования (главы 8 и 9)	Тестовые процедуры и тестовые варианты

Главы 3 и 4 посвящены анализу требований и объясняют, как составить требования, определяющие, каким должен быть продукт. Хотя задача написания спецификации выглядит весьма простой, на практике сложно выполнить ее хорошо. Некоторые требования описываются наилучшим образом с использованием формальных математических методов. В главе 5 объясняется, как описать архитектуру программного обеспечения, следуя классификации Гарлана и Шоу. В главе 6 описывается, как провести детальное проектирование, в том числе с использованием образцов проектирования — тех замечательных элементов «языка» проектирования, которые сильно облегчают нам обсуждение различных разработок. Глава 7 посвящена непосредственно разработке (программированию). Особое внимание в ней уделяется стандартам и формальным методам, помогающим разработчикам писать программы, правильность которых легко проверить. В главах 8 и 9 рассказывается, как тестировать приложение по частям и в целом. Там же идет речь об артефактах тестирования — в их число входят процедуры тестирования, описывающие процесс тестирования и примеры, содержащие тестовые входные параметры.

0.7. Качество

Приложения должны удовлетворять заранее определенному уровню качества. Для достижения требуемого уровня качества применяются следующие методы.

- *Инспектирование* (глава 1). Командный процесс обеспечения качества, применяется на всех стадиях проекта.
- *Формальные методы* (глава 1). К ним отсносятся математические методики для доказательства правильности программы, то есть того, что она делает то, что предполагается. Применяются выборочно.

- *Тестирование*:
  - на уровне модуля (компонента) (глава 8);
  - на уровне целого приложения (глава 9).
- *Методы управления проектом* (глава 2):
  - предсказание стоимости и сроков;
  - управление артефактами (версиями, документами и т. д.).

Крупные приложения, даже написанные выдающимися компаниями, содержат ошибки. Принятие этого факта может показаться капитуляцией, но это не так. Возьмем для сравнения строительную инженерию. Без сомнения, рассматривая мост Золотые ворота, мы обнаружим облупившуюся краску или подобные дефекты, следовательно, мост не совершенен. Но важно не это, а то, что мост соответствует особым стандартам качества, таким как способность выдерживать перевозку грузов с одного конца на другой. Так же обстоит дело и с программным обеспечением. Вопрос только в том, что является эквивалентом «облезшей краски» для приложений. Программное обеспечение не изнашивается наподобие вещей, следовательно, необходимо особое определение его качества.

Вместо того чтобы настаивать на совершенстве, мы настаиваем на соответствии стандартам качества. Это возлагает на нас обязательство точно, то есть численно, определить эти стандарты. Такие численные данные называются *метриками*. Примером метрики является «количество ошибок, обнаруженных за месяц работы» при заранее установленном уровне их серьезности. Как только метрики и их допустимые границы определены, необходимо удостовериться, что примененный процесс и выполненный проект отвечают установленным границам метрик. Мы включаем проверку качества в каждый этап разработки и выделяем три направления проверки: инспектирование, доказательство правильности и тестирование.

Инспектирование — это процесс проверки качества, ориентированный на команды разработчиков. Он применяется на всех этапах разработки. *Доказательство правильности* — это математическая или логическая методика, используемая для убеждения себя и других в том, что программа делает то, что должна делать. Такое доказательство является *формальным* (или строгим) методом. Мы не исполняем программы во время этого процесса, а изучаем их исходный код. С другой стороны, мы запускаем программы в процессе *тестирования*. Тестирование на уровне модулей (компонентов) рассматривается в главе 8. Глава 9 посвящена сборке программы (интеграции частей) и тестированию ее в целом.

Неискушенные разработчики программного обеспечения часто удивляются, как много времени и средств уходит на тестирование. Автор встречался со многими разработчиками, которым ужасно надоело тестирование и которые надеются на появление «другого способа». Этот «другой способ» заключается в улучшении процесса в совокупности с инспекциями и подходящими формальными методами.

Для обеспечения соответствия стандартам качества необходимо внимательное управление артефактами, которые порождает процесс разработки. Многие из

них выпускаются в различных версиях, более поздние из которых улучшают или расширяют предыдущие. Управление такими артефактами называется *управлением конфигурациями* и обсуждается в главе 1.

Обеспечение качества требует от нас контролирования всего проекта, что зачастую бывает очень трудно. В частности, мы должны правильно оценивать использованные ресурсы, потенциальные возможности и текущее состояние продукта по отношению к графику работы. Лучше, если мы сможем все это предвидеть на основании использования стандартных методов, применимых к разным проектам.

0.8. Проект для студенческой команды

Читатели этой книги должны по ходу чтения работать над своим программным проектом, а лучше всего участвовать в командном проекте. Для многих это первый технический курс, в котором идет речь о групповой работе. Многие студенты потом вспоминают свои студенческие проекты как приключения. Если группе удается избежать некоторых распространенных подводных камней, то это приключение может стать одним из самых полезных и познавательных впечатлений за все время обучения. Советы «Один из способов...», которые приводятся в этой книге, помогут группам максимизировать преимущества коллективной работы. В дополнение к этому пример, описанный далее в разделе 0.11, демонстрирует образцы проекта, рассчитанного на группу разработчиков. Ближе к концу каждой главы помещен пример, фактически являющийся путеводителем по студенческому проекту с пояснениями и показывающий, как предполагаемая команда занимается разработкой примера.

ОДИН ИЗ СПОСОБОВ РЕШИТЬ НАЧАЛЬНЫЕ ВОПРОСЫ В КОМАНДЕ

Чем раньше группа приступит к работе, тем лучше. На начальной стадии разработки должны быть выполнены следующие этапы.

1. Установите план собрания и временные рамки. (В главах 1 и 2 детально рассказывается о том, как это сделать.)
2. Выберите руководителя команды.
3. Решите, как члены команды будут общаться.
4. Определите заказчика (тех, кому нужно ваше приложение).
5. Придите к пониманию проекта в общем. (Если проект кажется вам слишком туманным, это не страшно. Спрашивайте, пока не почувствуете, что предмет вам достаточно понятен.)

Одной из распространенных причин неудачи является невыполнение рабочих обязательств одним или несколькими членами группы (см. пункт 1 выше). Гораздо лучше обсудить обязанности с самого начала, чем пытаться исправлять проблемы по ходу развития проекта. Чтобы избежать возможных трений оттого, что одни члены группы вкладывают в проект больше сил, чем другие, определите заранее количество часов в неделю, уделяемых проекту. Это также будет способствовать более эффективной работе каждого участника.

ОДИН ИЗ СПОСОБОВ ОПРЕДЕЛИТЬ ОЖИДАНИЯ УЧАСТНИКОВ КОМАНДЫ

Вопросы, которые необходимо разобрать на первом собрании, перечислены ниже. Это поможет избежать будущих проблем.

1. Получите согласие всех участников на то, что они будут посвящать определенное количество времени работе над проектом:
   - определите ожидаемое количество часов в неделю;
   - если не удается добиться согласия, то в случае промышленного проекта поговорите с руководством, а в случае учебного проекта проинформируйте преподавателя и проведите взаимную письменную оценку;
   - соберите данные об известных датах отсутствия кого-либо из команды.
2. Выберите основную цель проекта: успешная разработка или обучение.
   - Успех в разработке рабочего продукта дает хорошее представление о руководстве, общении с заказчиком.
   - Обучение иногда заставляет пожертвовать достижениями, чтобы позволить участникам проекта изучить новые области.
   - Необходимо учитывать цели вышестоящего руководителя (преподавателя).

Второй момент (см. пункт 2 выше) касается компромисса между желанием произвести впечатляющий продукт и желанием научиться чему-то новому. Здесь часто кроется противоречие. Для создания мощного продукта каждый член группы должен заниматься тем, в чем он имеет наибольший опыт. Это типично производственный подход. Напротив, чтобы приобрести новые знания, необходимо заниматься тем, в чем меньше всего опыта (например, в управлении коллективом). Группа должна найти баланс между этими двумя целями и определить, когда нужно применять имеющийся опыт, а когда можно попробовать что-то новое. Инструкторы вознаграждают учебную активность участников, устанавливая особые критерии оценки в дополнение к оценке возможностей производимого продукта. Мы будем разбирать эти вопросы более подробно при обсуждении графика работы и обязанностей в главах 1 и 2. На начальной стадии группа может прийти только к общему пониманию, например:

Мы выберем как минимум четырех человек для участия в каждом этапе процесса. Один из этих четверых должен быть самым опытным из группы и иметь наибольшие знания об этом этапе.

0.8.1. Общение в группе

В самом начале проекта необходимо выбрать способы коммуникации участников группы и опробовать их. При этом лучше всего записать все выбранные аспекты. Пример организации общения в группе представлен ниже.

При несоблюдении правил общения могут возникнуть многочисленные проблемы. Чтобы избежать пробелов в понимании, нелишним будет посылать письмо сразу нескольким адресатам, заинтересованным в предмете, а не только основному собеседнику. Убедитесь, что вас правильно поняли. Проверьте собственное понимание, задавая дополнительные вопросы.

ОДИН ИЗ СПОСОБОВ ОРГАНИЗОВАТЬ ОБЩЕНИЕ В ГРУППЕ

1. Общая политика: если сомневаемся — обсуждаем. Многословие — не проблема.
2. Собрания: группа собирается каждый четверг с 8:00 до 10:00 в комнате 671, если не было дополнительной информации об изменениях.
3. Дополнительные собрания: члены группы оставляют время в понедельник с 16:00 до 18:00 свободным на случай дополнительного собрания.
4. Стандарты: в качестве текстового редактора будет использоваться программа Ajax версии 9. Программа электронной почты — BestMail 4, а если это приложение недоступно, то проверить имеющуюся почту на совместимость, особенно это касается присоединенных файлов.
5. Предпочтительный способ электронных коммуникаций: информация, представляющая интерес для всех членов группы, должна публиковаться на сайте группы (http://www.xxx.yyy) с автоматическим уведомлением всех членов группы. Формат темы должен выглядеть следующим образом: «Вниманию <имя или имена>: тема письма».
6. Альтернативный способ электронных коммуникаций: для персонального общения, не представляющего большого интереса для группы, использовать электронную почту или телефон.
7. Подтверждения: члены группы должны подтверждать получение электронных писем, адресованных им, вне зависимости от наличия просьбы подтвердить получение. Отправитель не должен оставлять без внимания случаи, когда важное письмо не получает подтверждения.

Если группа состоит из студентов, живущих по соседству друг с другом, или из инженеров, работающих в одном здании, то общение, как правило, протекает гладко. Однако даже в этом случае часто возникают пробелы в понимании, так что не стоит ожидать адекватного общения без процедуры, подобной той, что показана выше. Члены многих групп, в том числе и студенческих, находятся далеко друг от друга, и правильно построенное общение для них очень важно. Большие проекты часто ведутся несколькими командами, находящимися в разных частях страны. Слияние компаний, их покупка, образование совместных предприятий также нередко приводят к тому, что несколько групп работают над одним проектом.

Выпускается все большее количество продуктов, способных облегчить работу групп, таких как программное обеспечение коллективного пользования (groupware), видеоконференции и системы моментальных сообщений (instant messaging). Несмотря на все чудеса техники, нельзя переоценить значение правильно организованной личной встречи.

Групповой проект — это приключение. В добрый путь!

0.9. Обзор учебного проекта

Для обобщения и закрепления концепций, представленных в этой книге, по ходу книги разрабатывается пример проекта — компьютерная игра *Встреча*.

Помимо того, что компьютерные игры обладают развлекательным аспектом, они принадлежат к наиболее продаваемым программным продуктам (по сведениям журнала «Economist», 1998 г.) и связаны с серьезным бизнесом. Некоторые

прогнозы обещают увеличение количества игроков в компьютерные игры до 32 миллионов к 2002 году («The Boston Globe», 21.01.00, стр. D12). Предположим, что принято решение создать новую ролевую игру, способную заинтересовать всех взрослых независимо от пола. Увы, игра *Встреча* еще недостаточно хороша, чтобы в нее играть (у нас просто не было достаточно средств на ее разработку), но она служит хорошим примером к тексту этой книги, показывающему, как разрабатываются программные продукты, начиная от возникновения концепции и до введения в эксплуатацию и стадии поддержки.

Данный раздел содержит краткий обзор примера. Поскольку наш пример является для студентов образцом для подражания, мы постарались дать студенческим командам представление о возможностях дальнейшего развития таких проектов.

0.9.1. Компьютерная игра Встреча: введение

В игре *Встреча* представлена целиком или частично жизнь главного героя. Успешность в игре измеряется числом очков-жизней, набранных игроком, или способностью героя прожить максимально долгую жизнь. Типичная картинка из игры показана на рис. 0.2: персонаж, управляемый игроком (справа), и внешний персонаж игры (слева), находящиеся в зоне «двор».

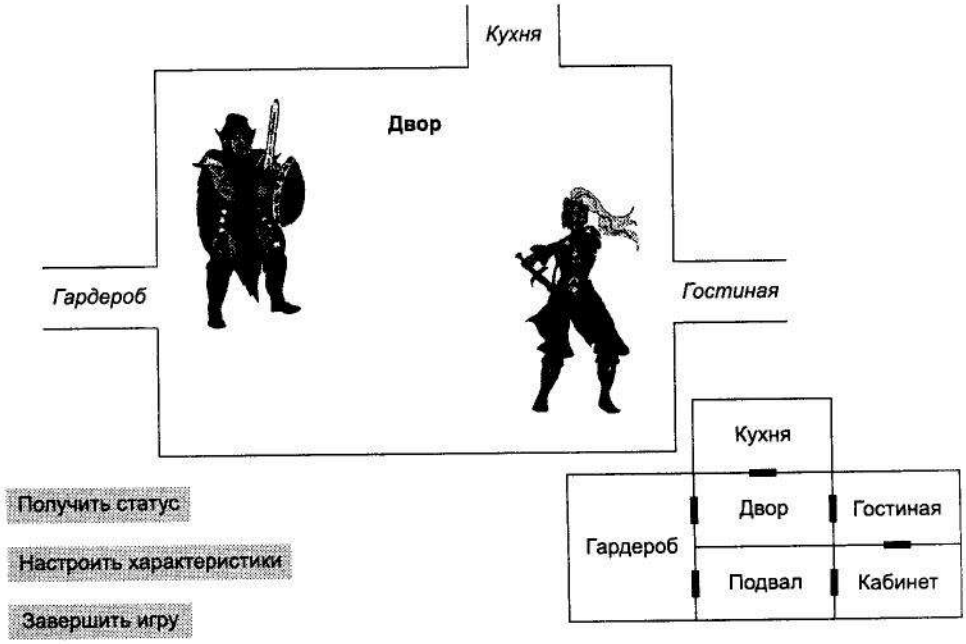


Рис. 0.2. Пример экрана из игры Встреча

Герои игры имеют определенное количество очков-жизней, распределенных по их характеристикам, таким как *сила*, *выносливость*, *терпение* и т. д. Персонажи

взаимодействуют друг с другом, когда они одновременно находятся в одной и той же зоне. Результат взаимодействия зависит от уровня способностей персонажей и от окружения, в котором они находятся. Контакт не обязательно протекает как борьба или соперничество. После завершения контакта герой случайным образом перемещается в другую зону игрового пространства.

Интерфейс для установки характеристик персонажа игры *Встреча* показан на рис. 0.3. Игрок может установить уровни характеристик героя в любое время, кроме момента непосредственного контакта с внешним персонажем. Одной из особенностей игры является то, что новый уровень характеристик активизируется не сразу, а персонаж на некоторое время остается уязвимым.

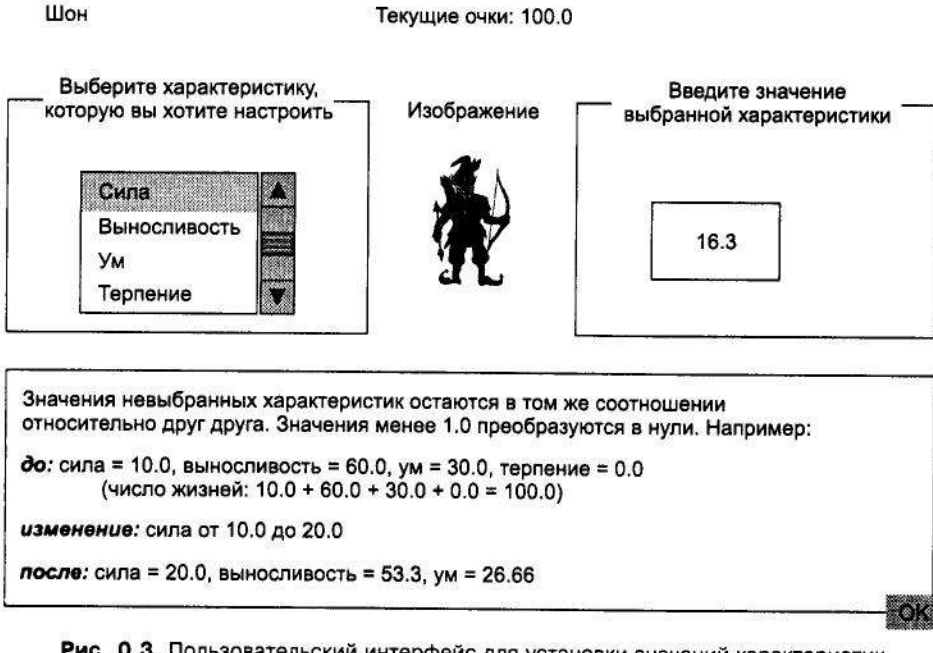


Рис. 0.3. Пользовательский интерфейс для установки значений характеристик

0.9.2. Требования к игре Встреча

Многие требования к игре *Встреча* могут быть выражены через взаимодействия между приложением и внешней стороной, обычно пользователем. Эти взаимодействия называются *вариантами использования* [63] и объясняются в главе 3. Например, вариант «Контакт с внешним персонажем» показан на рис. 0.4. «Контакт с внешним персонажем» — это последовательность действий, происходящая, когда герой и внешний персонаж находятся в одной зоне в одно и то же время. Подробные требования для *Встречи* могут быть изложены различными способами, например с помощью *диаграмм последовательности*, обсуждаемых в главе 4. Они являются графическим представлением управляющей логики программы, особенно удобным для визуализации вариантов использования.

Рис. 0.4. Вариант использования «Вступить в контакт с внешним персонажем»

0.9.3. Проектирование игры Встреча

Архитектура *Встречи*, представленная в главе 5, раскладывается на части, показанные на рис. 0.5. Вся архитектура разбивается на две части: *уровень ролевой игры* и *уровень игры Встреча*. Уровень ролевой игры состоит из следующих модулей-пакетов: *ПерсонажиИгры*, *РолеваяИгра* и *ИгроваяСреда*, а уровень компьютерной игры *Встреча* — *ПерсонажиВстречи*, *ИграВстреча* и *СредаВстречи*. Как будет объясняться в главе 5, уровень ролевой игры является *каркасом*, который можно применять для многих других игр.

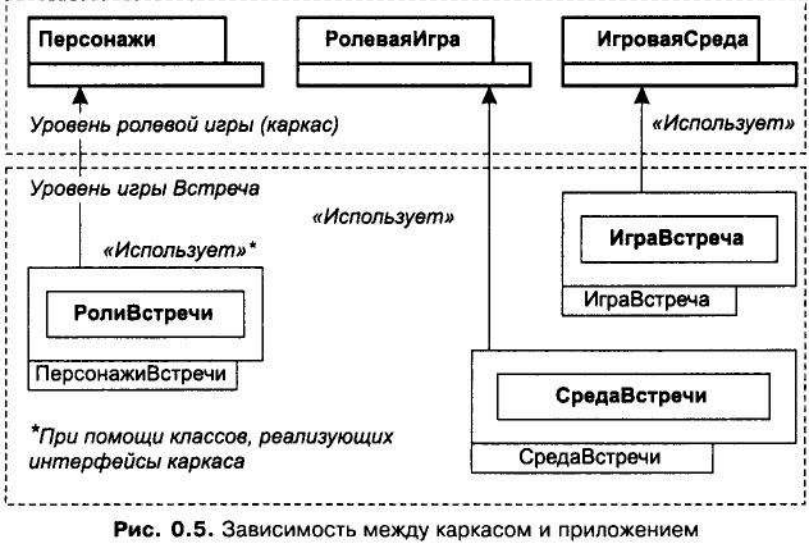


Рис. 0.5. Зависимость между каркасом и приложением

Пакет *РолеваяИгра* управляет перемещениями элементов в ролевой игре. Пакет *ПерсонажиИгры* включает как персонажей, управляемых игроком, так и внешних, находящихся под контролем приложения. Пакет *ИгроваяСреда* поддерживает игровое пространство.

Пакет *ИграВстреча* состоит из классов, контролирующих ход игры в целом. Пакет *ПерсонажиВстречи* включает персонажей игры, как управляемых игроком, так и прочих. Пакет *СредаВстречи* описывает пространство игры *Встреча*, в том числе отдельные зоны и соединения между ними.

0.9.4. Тестирование игры Встреча

Тестирование игры *Встреча* состоит из модульного, интегрального и системного тестов. Модульное тестирование включает тестирование отдельных функций класса, а затем их комбинаций. Следующий шаг — это интегральное тестирование, в процессе которого проверяется общая функциональность приложения на каждой стадии сборки. И, наконец, системное тестирование состоит из тестирования конечного продукта. В главах 7 и 8 обсуждаются эти и некоторые другие виды тестирования.

0.9.5. Документация по проекту видеоигры Встреча

Проектная документация приблизительно соотносится с водопадным процессом разработки. В примере используются ключевые документы, описанные в стандартах IEEE. *План контроля качества ПО* для игры *Встреча*, приведенный в конце глав 1 и 2, устанавливает проектную документацию, стандарты, проверки и управление рисками. *План управления конфигурациями ПО*, приведенный в конце главы 1, показывает, как должны храниться коды и документы. *План управления программным проектом*, находящийся в конце главы 2, показывает подход к управлению проектом. *Спецификация требований к ПО*, приведенная в конце глав 3 и 4, описывает требования, предъявляемые к приложению. *Проектная документация ПО*, расположенная в конце глав 5 и 6, демонстрирует архитектуру и подробную разработку приложения. *Документация по тестированию ПО*, приведенная в конце глав 8 и 9, описывает способы, которыми тестировалось приложение.

Упражнения

Ответы и подсказки для упражнений, помеченных символами «о» или «п», приводятся в конце этой главы.

П1[о].

1. Что такое *инженерная наука*? Дайте краткое определение.
2. Чем *инженерное дело* отличается от *науки*? Приведите ключевые характеристики, отличающие одно от другого.

П2°. Каковы четыре основные «П», составляющие технологию разработки программного обеспечения?

Упражнения в команде

Обсудите в группе, как вы будете выполнять следующие упражнения (прежде чем выполнять задание, обратите внимание на подсказки, приведенные ниже).

К1". Решите, кто будет лидером вашей группы. Заметьте, что будучи лидером, вы сможете приобрести опыт, который в противном случае будет тяжело получить.

К2". Решите, как члены команды будут общаться между собой, определите средства и методы коммуникаций, протестируйте методы. Опишите все подробно, позднее вы сможете изменить детали.

К3". Найдите в Сети новейшую информацию по теме, заданной руководителем (например, о командном процессе разработки программного обеспечения). Отметьте по меньшей мере четыре основные цели и пять используемых методов. Пошлите ссылки от имени группы на курсовой форум или веб-сайт, если таковой есть. Сформулируйте индивидуальное или коллективное мнение по данному вопросу.

Отчет вашей команды должен быть около 4–7 страниц.

Критерии оценки:

- понятность («Отлично» — очень понятно написано, все важные моменты объяснены, не слишком многословно);
- конкретность («Отлично» — приведены конкретные процедуры, описывающие общение в группе при всех наиболее вероятных обстоятельствах);
- правильность изложения темы («Отлично» — весьма ясно, что писавший понимает цели изучения данной темы; изложение правильно организовано).

Ответы

П1.

1. Определение «инженерной науки» — в первом предложении введения. Наука — это область человеческой деятельности, которая расширяет наши знания и понимание с помощью строгих проверяемых методов.
2. Наука вообще и инженерная наука пересекаются, когда появление новых знаний приводит к решению одной из общечеловеческих проблем.

П2. Процесс, проект, персонал и продукт.

Подсказки

К1. Для того чтобы многие могли попробовать себя в качестве руководителя, целесообразно поменять лидера группы примерно в середине семестра. Оба лидера могут быть выбраны в начале семестра и заменять друг друга в случае необходимости. Такой подход хорош еще и потому, что лидер группы может внезапно перестать

заниматься проектом или вообще посещать занятия. Заметьте, что во время выполнения второй половины проекта от лидера требуется более быстрое принятие решений, чем во время первой.

К2. Для примера возьмем телефонные звонки, собрания группы, электронную почту, форумы, чаты и веб-сайты.

1. Запланируйте регулярные собрания группы по крайней мере раз в неделю. Трудно организовать незапланированную встречу, зато легко отменить запланированную.
2. Электронная почта — это важное средство общения, но она может оказаться не слишком удобной из-за непредвиденных задержек. Синхронизация сообщений может быть нарушена, что приведет к потере нити диалога. Это становится особенно важным ближе к концу проекта, когда необходимо частое общение, от которого может зависеть вся работа.
3. Используйте общий веб-сайт или чат. Например, бесплатные услуги доступны на [20].
4. Недостаточно определить: «Мы будем использовать программу *Superword* для написания текстов». Обязательно определите номер версии и обменяйтесь для верности несколькими сообщениями. Не меняйте версию по ходу проекта, не проверив ее совместимость с предыдущей.
5. Опробуйте все выбранные методы и стандарты.

(Во время действия этой учебной программы проверяйте свои планы и намерения практическим тестированием. Постарайтесь провести по крайней мере два независимых тестирования. В целом, знайте, что ваш проект будет приобретать все большее значение по ходу его развития.)

К3. Аналогичным образом проверьте вашу систему коммуникаций. Например, вы можете создать веб-сайт, куда члены группы будут помещать новую информацию о TSP. Как вы упорядочите эту активность? Как получить полезный результат вместо конгломерата несвязанных текстов?

Глава 1
Процесс

...Пройдут века,
И в странах, что еще не существуют,
Актеры будут представлять наш подвиг.
Шекспир. Юлий Цезарь

Хорошие разработчики программного обеспечения избегают повторения ошибок предыдущих проектов за счет документирования и совершенствования своего процесса разработки.

Используя в качестве основы водопадную модель процесса разработки программ, настоящая глава освещает вопросы, перечисленные на рис. 1.1. Основные цели этой главы следующие.

- Провести различие между процессами разработки:
  - определить их достоинства и недостатки.
- Количественно определить *качество* программного обеспечения.
- Понять необходимую документацию:
  - примерно по одному документу для каждой фазы водопадного процесса;
  - план для управления конфигурациями.

Главы 1, 2, 4, 5 и 6 разбиты на две части — «Основы» и «Детали». При первом прочтении каждую из частей «Детали» можно пропускать, откладывая их изучение на потом. Прохождение материала в таком порядке позволит студенческим командам разработать небольшой прототип их учебного проекта — реализацию избранных минимальных аспектов приложения, как объясняется в разделе 5 главы 3. Этот прототип может быть использован позже в ходе полного анализа требований (главы 3 и 4) и процесса проектирования (главы 5 и 6). Такой прототип должен быть максимально простым. Его основное предназначение — позволить команде попрактиковаться в совместной работе над проектом.

- Основы: разделы 1.1–1.5.
- Детали: разделы 1.6–1.9.
- Упражнения.
- Примеры:
  1) План управления конфигурациями программного обеспечения (SCMP);
  2) План контроля качества программного обеспечения (SQAP), часть 1 (часть 2 находится в конце главы 2).

Рис. 1.1. Схема разработки программ: темы главы

Основы

1.1. Введение в процесс разработки программного обеспечения

Проектирование программного обеспечения представляет собой процесс построения приложений реальных размеров и практической значимости, удовлетворяющих заданным требованиям функциональности и производительности, таких, например, как текстовый редактор, электронная таблица, операционная система или, скажем, программа контроля неисправностей космической станции. *Программирование* — это один из видов деятельности, входящих в цикл разработки программного обеспечения.

По масштабам работы, требуемым профессиональным знаниям и общественной значимости различие между просто программированием и проектированием про-

граммного обеспечения можно сравнить с различием между изготовлением скамейки у ворот своего загородного дома и возведением моста. Эти две задачи различаются на порядок по значимости и требуемым профессиональным знаниям. В отличие от постройки скамейки возведение моста включает в себя как профессиональную, так и социальную ответственность. Хотя социальная сторона вопроса оставлена за рамками этой книги, мы все же рассмотрим связанные с ней технологии, такие как строгий анализ требований и стандарты количественной оценки качества.

Технология разработки программного обеспечения должна охватывать разнообразные типы программ, включая перечисленные ниже.

- Автономное:
  - устанавливаемое на одиночный компьютер;
  - не связанное с другим программным и аппаратным обеспечением;
  - пример — текстовый редактор.
- Встроенное:
  - часть уникального приложения с привлечением аппаратного обеспечения;
  - пример — автомобильный контроллер.
- Реального времени:
  - должны выполнять функции в течение малого интервала времени, обычно нескольких микросекунд;
  - пример — программное обеспечение радиолокатора.
- Сетевое:
  - состоит из частей, взаимодействующих через сеть;
  - пример — основанная на веб-технологии видеоигра.

Излагаемые в этой книге принципы применимы ко всем этим типам. Отметим, однако, что разработка встроенных программ и программ реального времени имеет дополнительные аспекты, анализ которых выходит за рамки нашего исследования.

1.1.1. Типичная схема разработки программного обеспечения

Как же рождаются сложные и пригодные к использованию приложения? Стандартная последовательность шагов такова.

1. Понять природу и сферу применения предлагаемого продукта.
2. Выбрать процесс разработки и создать план.
   Раздел 1.4 и глава 2.
3. Собрать требования.
   Главы 3 и 4.
4. Спроектировать и собрать продукт.
   Главы 5, 6 и 7.

5. Выполнить тестирование продукта.

   Главы 8 и 9.

6. Выпустить продукт и обеспечить его сопровождение.

   Глава 10.

1. Прежде всего, необходимо уяснить суть проекта. Это кажется очевидным, однако для того, чтобы понять, чего хочет заказчик, требуется ощутимое время, особенно если заказчик сам не знает достаточно хорошо, чего он в действительности хочет. Нужно составить представление о масштабах проекта и с этой целью оценить, какими сроками, финансами и персоналом мы располагаем. Если, например, для построения видеоигры нам предоставляется 10 тыс. долларов, один разработчик и один месяц срока, можно говорить разве что о реализации прототипа игры (что и делается в примере из данной книги). Если же мы располагаем бюджетом в 5 млн долларов, 20 разработчиками и сроком в 18 месяцев, то речь уже может идти о создании полноценного конкурентоспособного программного продукта и совсем других масштабах производственной деятельности.
2. С самого начала проекта должна вестись документация, которая, скорее всего, будет изменяться и обновляться в ходе разработки. Поэтому сразу же необходимо определиться и со средствами, при помощи которых будут отслеживаться вносимые в документацию и в программный код изменения. Этот процесс называется *управлением конфигурациями* и рассматривается в разделе 1.7.3. Управление конфигурациями не является абсолютной необходимостью, но его отсутствие может привести к большим недоразумениям и общей потере продуктивности. Далее нужно определиться с самим процессом разработки. Разновидности методов разработки рассматриваются в разделе 1.4. Иногда стандарты, принятые внутри компании, определяют используемый процесс.

   После этого обычно составляется общий план проекта, включающий в себя план-график (расписание проекта). Этот план будет уточняться на протяжении жизненного цикла проекта, по мере того как будут уточняться требования к проекту и детали реализации. Например, детали плана-графика не могут быть проработаны, пока не будет определена архитектура приложения.
3. Следующий шаг состоит в сборе требований к приложению. Он включает в себя, прежде всего, обсуждение проекта с заказчиками и другими участниками, заинтересованными в его выполнении. Процесс сбора требований детально рассматривается в главах 3 и 4.
4. Затем наступает черед проектирования и реализации самого продукта. Этим вопросам посвящены главы 5, 6 и 7.

   В зависимости от используемого процесса разработки шаги 3 и 4 могут быть выполнены несколько раз.
5. Программный продукт, как окончательный, так и промежуточный, подлежит тщательному тестированию. Тестированию посвящены в главы 8 и 9.

6. Наконец, когда продукт выпущен, наступает фаза его сопровождения, включающая внесение в него исправлений и улучшений. Сопровождение, которое требует до 80 % ресурсов, потребовавшихся на разработку, рассматривается в главе 10.

1.2. Исторический и современный взгляд на разработку программного обеспечения

Настоящий раздел содержит краткий исторический очерк развития технологии разработки программного обеспечения и обзор современных направлений и методов.

1.2.1. Становление инженерии программного обеспечения

Разработка программного обеспечения является очень молодой и быстро развивающейся отраслью инженерной науки. Она подвержена постоянным и быстрым изменениям. Так, всего лишь в начале 90-х годов Британское сообщество вычислительной техники (British Computer Society) начало присваивать разработчикам программ звание инженера (Chartered Engineer), а в Соединенных Штатах только в 1998 году стало возможным хоть где-то (а точнее, в штате Техас) зарегистрироваться в качестве профессионального инженера программного обеспечения. Но по-прежнему, даже в начале двадцать первого века, общепризнанным остается тот факт, что разработке программного обеспечения не достает достаточно развитой научной базы. По некоторым оценкам, 75 % организаций, занимающихся разработкой программ, делают это на примитивном уровне. С другой стороны, в этой области сформировалось немало интересных идей, и мы надеемся, что знакомство с ними в рамках настоящей книги вдохновит читателя на собственное исследование.

1.2.2. Влияние структурного и объектно-ориентированного программирования

С момента зарождения технология разработки программ испытала несколько подъемов в своем развитии. Один из них связан с публикацией письма Эдсгера Дийкстры (Edsger Dijkstra) в Ассоциацию вычислительной техники (ACM — Association for Computing Machinery), озаглавленного так: «О вреде использования операторов GOTO» [26]. В те времена программы писались с активным использованием операторов безусловного перехода. Обращая внимание на недостатки таких программ, Дийкстра предложил концепцию *структурного программирования*, позволяющую избежать использования этих операторов. Концепция Дийкстры основывалась на том наблюдении Бема и Якопини [12], что для записи

любой программы в принципе достаточно только трех конструкций управления — последовательного выполнения, ветвления и цикла, то есть что теоретически необходимость в использовании операторов перехода отсутствует.

Следующий шаг в развитии структурного программирования связан с введением аппарата функций, позволяющих разбивать структурную программу на обозримые по своим размерам части. При таком подходе программа пишется в терминах вызова функций верхнего уровня, которые реализуются при помощи функций более низкого уровня, и т. д. Этот процесс нисходящего программирования иллюстрирует рис. 1.2.

Рис. 1.2. Структурное программирование

Нисходящее структурное программирование стало само собой разумеющимся стилем написания программ, и без него вряд ли был бы возможен прогресс в области разработки программного обеспечения. Однако структурным программам в таком виде недоставало одного важного свойства — в их структуре непосредственно не отображались сущности предметной области, и из-за этого программы было трудно модифицировать в условиях изменяющихся требований. Позднее в связи с этим возникла парадигма *объектной ориентированности* (ОО), которая основана на использовании объектов, объединяющих в себе данные и функциональность. Объектно-ориентированную парадигму иллюстрирует рис. 1.3 на примере отображения в программе понятий «заказчик» и «счет». Помещая каждый функциональный элемент в соответствующий класс, мы значительно облегчаем процесс проектирования и сопровождения программ.

На объектно-ориентированной парадигме основаны современные языки и системы программирования, такие как Java и CORBA. Отметим, что CORBA позволяет приложениям использовать функции, написанные на различных языках, и выполнять их на различных платформах. В качестве примера в этом же контексте можно упомянуть Visual Basic и рассматриваемую в следующем разделе модель COM фирмы Microsoft.

В этой книге предполагается, что читатель владеет методами объектно-ориентированного программирования.

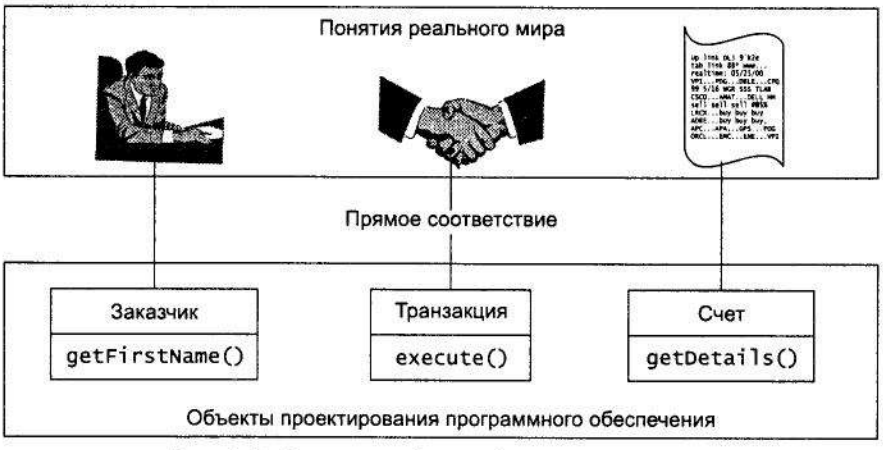


Рис. 1.3. Парадигма объектной ориентированности

1.2.3. Повторное использование компонентов

Говорят, что Генри Форд совершил революцию в производстве автомобилей, когда заметил, что узлы автомобиля можно стандартизировать, так что при сборке автомобилей данной модели можно будет использовать любой экземпляр требуемого узла. Это удешевило процесс сборки и сделало автомобили доступными по цене широким слоям населения.

Столь же важной в настоящее время признается возможность при разработке одних приложений заимствовать идеи, архитектуру, проект и исходный код других приложений. Если приложения проектируются таким образом, что различные их части (обычно это какие-либо наборы классов) могут быть использованы многократно, то в конечном итоге это приводит к уменьшению стоимости разработки приложений. Однако чтобы это было возможно, приложения должны быть *модульными*. Модульность приложения, собственно, и означает, что оно состоит из легко идентифицируемых и заменяемых частей. По этой причине при правильном проектировании программного продукта особое внимание должно уделяться модульности, особенно на стадии разработки архитектуры (глава 5). Модульный подход также важен на этапах детального проектирования и реализации.

На уровне проектирования и реализации повторное использование программных кодов упрощается благодаря появлению таких стандартов, как COM (Component Object Model) и Java Beans. COM-объекты являются примером инструмента модульного программирования. Идею COM-объектов иллюстрирует рис. 1.4. COM-объект *Счет* — это фактически исполняемый модуль среды Windows, который может быть динамически использован любым другим приложением Windows. Этот объект поддерживает три интерфейса. В частности, он поддерживает интерфейс *Identification*, то есть в этом объекте имеется, например, функция `setName(...)` и т. п. Основная идея состоит в том, что объект *Счет* может быть

использован сколь угодно часто, так как мы знаем, для чего он предназначен. Заметим, что одни COM-объекты могут строиться из других COM-объектов и могут представлять довольно большие сущности, как, например, целая электронная таблица.

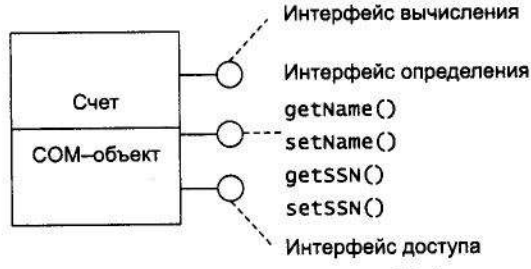


Рис. 1.4. Идея технологии COM

На общем уровне, согласно [80], программный компонент определяется как «элемент программы, обладающий следующими двумя свойствами:

- этот элемент может использоваться другими элементами программы, которые при этом называются *клиентами*;
- автору этого элемента нет необходимости знать клиенты и их авторов».

Хорошим источником сведений по общей концепции программных компонентов является [105].

1.2.4. Формальные методы

К *формальным* относятся те методы проектирования программ, которые основаны на математике. В свое время Дийкстра [25], Хоар [44] и другие указывали, что поскольку программы имеют точно определенное поведение, они могут рассматриваться как математические объекты. При этом полезно вспомнить, что развитие многих других инженерных дисциплин происходило именно на математической основе. Так, например, электротехника имеет глубокие корни в математике, и последняя в настоящее время является неотъемлемой частью электротехнического образования. Многие утверждают, что нечто аналогичное должно произойти и с разработкой программ.

Используя понятный и точный язык математики, формальные методы могут помочь решить задачу обеспечения надежности программ. Они могут быть применены как при анализе требований для обеспечения точности формулировки требований, так и в процессе реализации для обеспечения соответствия кода программы сформулированным требованиям. Формальные методы обсуждаются в главах 4 и 7.

Как правило, формальные методы используют математику в ее *логическом* аспекте. В своем же *вычислительном* аспекте математика задействована в связи с использованием метрик, которые рассматриваются далее.

1.2.5. Удобство и простота использования

Важной составляющей разработки программ является проектирование взаимодействия пользователя с программой. В частности, широкое применение графического пользовательского интерфейса ведет к тому, что многие аспекты разработки приложений лежат за пределами математики и даже алгоритмики (например, [109]) и, в частности, требуют серьезного внимания к психологическим основам взаимодействия человек — машина. В настоящей книге вопросы пригодности к использованию рассматриваются как составная часть анализа требований (главы 3 и 4), проектирования (главы 5 и 6) и тестирования (глава 9).

1.3. Требования к процессу, проекту, продукту и персоналу

Вспомним рассмотренные во введении четыре «П» разработки программ. Цель любого программного *проекта* состоит в производстве некоторого программного *продукта* (например, текстового редактора). То, как в рамках проекта производится продукт, представляет собой *процесс*. Поскольку критичным для успеха дела является взаимодействие членов команды, мы включаем в рассмотрение *персонал*.

Эта книга посвящена прежде всего процессу, проекту и продукту. Вопросы, касающиеся персонала, рассматриваются лишь частично и ограничиваются в основном затрагиваемыми в главе 2 ключевыми факторами, обусловливающими успешность проекта. Для получения подробных сведений по этим вопросам мы рекомендуем книги Брукса [18] и Хэмфри [54].

В современной практике разработки программного обеспечения существует пять ключевых требований, в основном сформулированных Хэмфри [50] (рис. 1.5). Первое из них — заранее выбрать свою шкалу измерения качества для проекта и продукта. Например, «500 строк полностью протестированного программного кода, написанного одним человеком за месяц» или «не более трех ошибок на тысячу строк программного кода». Второе требование состоит в сборе информации по всем проектам с целью создания базы для оценки будущих проектов. Третье положение гласит, что все требования, схемы, программные коды и материалы тестирования должны быть легко доступны всем членам команды. Суть четвертого условия состоит в том, что все участники команды должны следовать избранному процессу разработки[1]. Роль четырех «П» в достижении пяти ключевых требований демонстрирует рис. 1.5. При этом подчеркивается тот факт, что ничто не может быть достигнуто без участия членов команды, работающей над проектом. Например, создание программ только согласно выбранным решениям (пункт 4, *б* на рис. 1.5) — это требование, которое может быть проконтролировано в *проектах* и которое потребует определенных усилий от *персонала*, работающего над проектом.

[1] Пятое требование состоит в том, что выбранные показатели качества должны постоянно измеряться и эти измерения должны протоколироваться. В оригинале данное требование опущено. — *Примеч. науч. ред.*

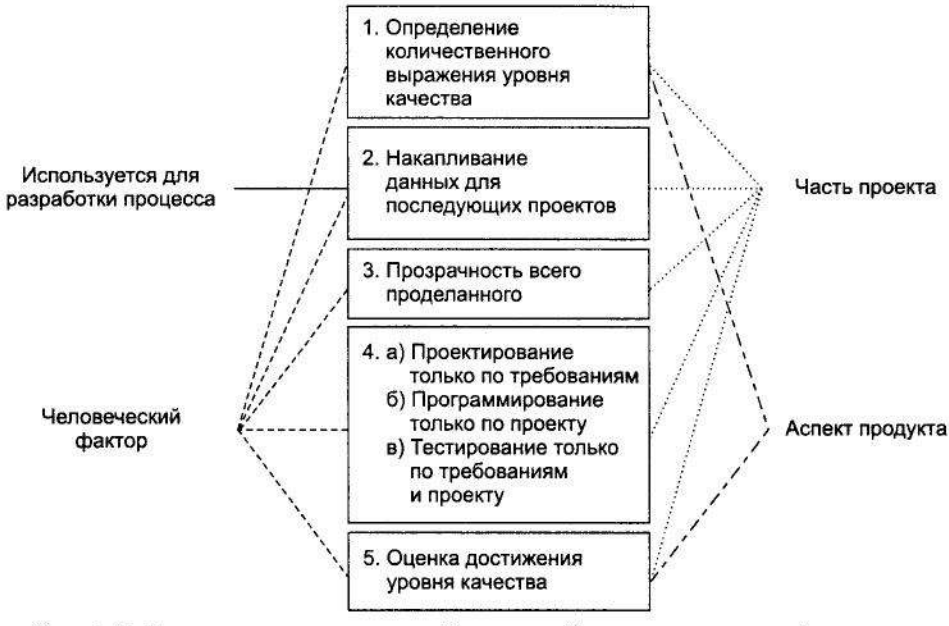


Рис. 1.5. Пять ключевых рекомендаций для разработки программного обеспечения

Достижение конкретных, заранее установленных и измеримых целей — вот главное требование к проекту и к конечному *продукту*. Все пять требований имеют свое преломление для конкретного *проекта*, и все пять зависят от деятельности *людей*.

1.3.1. Артефакты и роли

В понятие продукта разработки включается не только программный код — сюда относятся различные *артефакты*, такие как планы, отчеты, диаграммы. Кроме того, участвующие в процессе разработчики играют разнообразные *роли*. В терминологии рассматриваемого далее Унифицированного процесса разработки программного обеспечения (USDP) [64] эти роли называются *работниками* (workers). Используемая в USDP нотация для артефактов и работников приводится на рис. 1.6. В главах 5 и 6, посвященных программному проектированию, мы рассмотрим так называемые *модели*, позволяющие описывать проектирование с различных точек зрения.

1.4. Разновидности процесса разработки

Трудности конструирования реальных приложений обусловлены их сложностью, и критическую роль в преодолении этой сложности играет сам процесс конструирования. Существует несколько разновидностей процесса, и главная из них — это *водопадная модель*.

Рис. 1.6. Артефакты и роли

1.4.1. Водопадная модель процесса

Классической моделью процесса разработки программ является *водопадная модель*, в рамках которой процесс представляется последовательностью фаз *анализа требований*, *проектирования*, *реализации*, *интеграции* и *тестирования* (рис. 1.7).

- *Анализ требований* состоит в сборе требований к продукту. Результатом анализа, как правило, является некоторый текст. Эта стадия рассматривается в главах 3 и 4.
- *Проектирование* описывает внутреннюю структуру продукта. Обычно такое описание дается в форме диаграмм и текстов. Этот этап рассматривается в главе 5 (архитектура) и в главе 6 (детальное проектирование).
- *Реализация* — это программирование. Результатом реализации является программный код всех уровней, будь то код, генерируемый высокоуровневой системой программирования, компилятором языка четвертого поколения или какой-либо другой. Фазе реализации посвящена глава 7.
- *Интеграция* — это процесс сборки всего продукта из отдельных частей. Интеграция и тестирование рассматриваются в главах 8 и 9.

В действительности перечисленные фазы не следуют строго последовательно друг за другом, а частично перекрываются. На практике любую из фаз можно начинать до того, как будет полностью завершена предыдущая.

Рис. 1.7. Водопадная модель процесса разработки

Иногда водопадный процесс расширяют (рис. 1.8) следующими дополнительными фазами.

- *Концептуальный анализ*, состоящий в определении общих принципов приложения и выполняемый в самом начале процесса (глава 3).
- *Объектно-ориентированный анализ*, состоящий в выделении ключевых классов (раздел 1.2.2 и глава 6) и выполняемый после анализа требований и до фазы проектирования.
- Фазы *модульного* и *системного тестирования*, на которых тестируются соответственно отдельные части приложения и все приложение как целое.
- *Сопровождение*, состоящее в модификации и внесении исправлений в приложение и осуществляемое в самом конце процесса (глава 10).

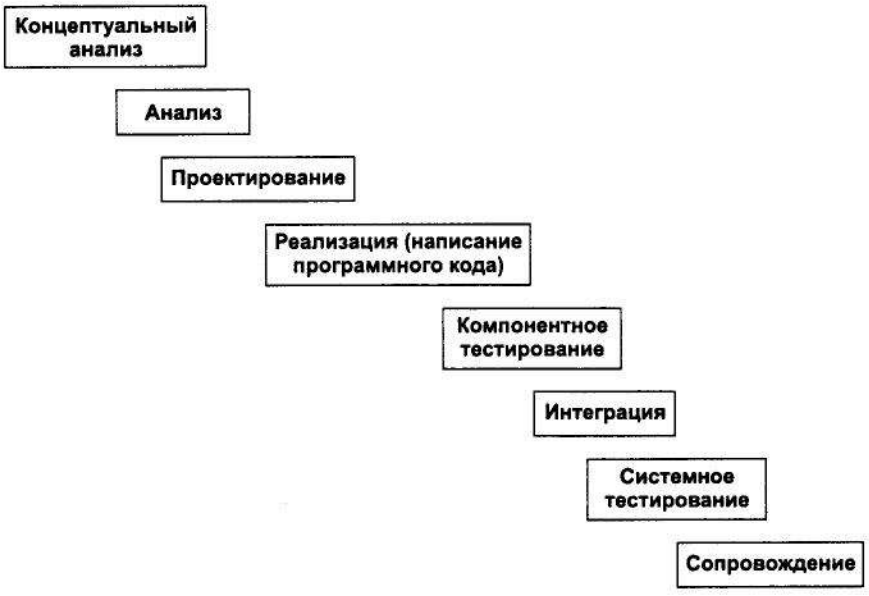


Рис. 1.8. Более детализированная модель водопадного процесса

В чистом виде водопадный процесс применяется достаточно редко, разве что в случае небольших проектов или когда команда реализует проект, очень похожий на один из тех, что были осуществлены ею ранее. Основной причиной неприменимости водопадного процесса в чистом виде является сложность большинства приложений. В нашей книге мы используем в качестве примера программного продукта ролевую игру, однако существует столь много различных толкований понятия «ролевая видеоигра» (RPG — Role-Playing Game), что было бы совершенно непрактично попытаться определить все до последнего требования к нашей игре, прежде чем приступать к проектированию и реализации. Тем не менее водопадный процесс является основой для большинства других разновидностей процесса.

Процессы, в которых водопадная схема применяется многократно, называются *итеративными*. Сразу оговоримся, что в итеративных процессах не обязательно все шаги водопадной схемы должны выполняться на каждой итерации. Ниже мы рассмотрим две разновидности итеративных процессов — *спиральные* и *инкрементальные* процессы.

1.4.2. Спиральная модель процесса

В случае *спирального* процесса последовательность *анализ требований — проектирование — реализация — тестирование* выполняется более одного раза. Для этого может быть несколько причин. Основная причина обычно связана с необходимостью предупреждения рисков (глава 2). Другой причиной может быть необходимость предоставить заказчику частичную версию проекта для получения отзывов и пожеланий. Если разрабатываемая программа достаточно сложна, необходимо выполнять промежуточные интеграции, не откладывая эту фазу на самый конец, как это предписывает водопадная модель. Общая же идея спирального процесса заключается в том, чтобы на каждой итерации строить очередную версию программы, используя в качестве основы ее предыдущую версию. В этом случае процесс приобретает спиралевидный характер (рис. 1.9).

Если взять в качестве примера нашу видеоигру, то первая итерация является не чем иным, как подготовкой персонажа, выбранного игроком, и его вступлением в игровую зону. Вторая итерация уже позволяет свободно передвигаться по схеме игры. На третьей итерации появляются соперники и союзники героя и т. д.

Дополнительное преимущество итеративных процессов заключается в возможности собирать на каждой итерации метрические характеристики процесса. Например, располагая данными о времени, которое потребовалось для выполнения первой итерации, мы можем уточнить план-график дальнейшей работы. Такая возможность особенно полезна для организаций, имеющих небольшой опыт планирования разработок.

Хотя спиральная модель отражает типичную схему процесса разработки, она требует более искусного управления, нежели простая водопадная модель. Одна из трудностей заключается в поддержке целостности документации, которая должна быть полностью обновлена и дополнена к концу каждой итерации. В частности, каждая версия программного кода должна реализовывать документированный проект и удовлетворять документированным же требованиям. Управление доку-

ментацией еще более усложняется, когда с целью повышения производительности команды очередная итерация процесса начинается до завершения предыдущей итерации.

Рис. 1.9. Спиральная разработка

И все же для большинства программных проектов преимущества спирального процесса перевешивают его недостатки. Здесь можно сослаться на опыт Министерства обороны США, которое, признав эти преимущества, в 80-х годах отказалось от принятой им ранее установки на использование простой водопадной модели во всех программных проектах.

Сколько же итераций требуется в случае применения спиральной модели? Это зависит от ситуации. Скажем, типичный проект, трудоемкость которого оценивается в три человеко-месяца, а продолжительность — в четыре месяца, вероятнее всего, потребует две-три итерации. Затраты на проведение большего числа шагов могут просто перевесить выгоду от дополнительных итераций. Далее, в разделе 1.4.4, мы покажем, на какие четыре группы разбиваются итерации в USDP.

Случай, когда число итераций возрастает настолько, что каждая новая итерация предоставляет слишком малое количество новых возможностей по сравнению с предыдущей, мы будем называть *инкрементальной* разработкой.

1.4.3. Инкрементальная модель процесса

Иногда представляется возможным понемногу продвигать проект вперед при практически непрерывном процессе. Такая модель процесса особенно полезна на поздних стадиях проекта, когда продукт находится на сопровождении или когда

разрабатываемый продукт очень схож с созданным ранее. Например, Кукумано и Сэлби [21] подготовили доклад по процессу, используемому в некоторых отделениях корпорации Microsoft, где обновления программного кода и документации представляются ежедневно к конкретному времени для интеграции и ночного тестирования. Другие организации используют для этого недельные циклы. Для поддержания соответствующего уровня инкрементальной разработки необходимо иметь четко установленную архитектуру проекта и исключительно синхронизированную систему документации (рис. 1.10). Для организации инкрементальной разработки обычно выбирается характерный временной интервал, например неделя. Затем в течение этого интервала происходит обновление исходного проекта (документации, набора тестов, программного кода и т. д.). Теоретически шаги разработки (increments) могут выполняться и параллельно, но такой процесс очень сложно скоординировать. Инкрементальная разработка проходит лучше всего, если следующая стадия $n+1$ начинается по возможности после того, как обновление всех модулей на стадии n закончено, и хуже всего, если время, требуемое на обновление модулей, значительно превышает выбранный интервал. Для того чтобы убедиться в этом, представьте, что необходимо изменить модуль 789, который зависит от семи других модулей: 890, 23, 489, 991, 7653, 2134 и 2314. Если изменение занимает девять недель, то модуль 789 должен быть построен исходя из предполагаемого состояния всех семи модулей через девять недель. Эту работу очень трудно скоординировать, так как каждая из семи частей может быть изменена до девяти раз (еженедельно), причем каждое новое изменение может основываться на исследовании эффективности предыдущих изменений.

[1] План управления программным проектом (SPMP, глава 2)

[2] Проектная документация программного обеспечения (SDD, глава 5)

[3] Спецификация требований к программному обеспечению (SRS, глава 3)

Рис. 1.10. Инкрементальная разработка

В отчете Кукумано и Сэлби указывается, что обычно Microsoft осуществляет разбиение проекта на части, затем применяет процесс инкрементальной разработки и синхронизации и периодически «стабилизирует» приложение путем сборки всех его частей. Они называют этот процесс «синхронизация и стабилизация».

1.4.4. Унифицированный процесс разработки программного обеспечения (USDP)

Унифицированный процесс разработки программного обеспечения (USDP – Unified Software Development Process) впервые был предложен в книге Якобсона, Буча и Рамбо [64], изданной в 1999 году. Этот процесс является детищем более ранних методологий, разработанных этими тремя авторами: «Объектная методология» Якобсона, «Методология Буча» [14] и «Техника объектного моделирования» Рамбо [96].

Поскольку итеративные подходы частично или полностью повторяют водопадный процесс, их иногда трудно описать, что доказывает рис. 1.10. USDP – это итеративный процесс, пытающийся разрешить эту проблему путем классификации итераций и отнесения их к одной из четырех групп.

- Начальные итерации – предварительные (подготовительные) взаимодействия с акционерами:
  - основной покупатель;
  - пользователи;
  - инвесторы;
  - другие.
- Итерации проектирования: их завершение желательно и просто необходимо; выбор базовой архитектуры.
- Итерации конструирования: приводят к изначальной оперативной способности.
- Итерации перехода: выпуск готового продукта.

Заинтересованные лица включают в себя всех, кто так или иначе заинтересован в реализации проекта. Это, конечно же, заказчики и пользователи (которые сами могут и не являться заказчиками), а также инвесторы, пользовательские группы и сами разработчики. Итерации *проектирования* задают ключевую техническую цель в выборе и утверждении (принятии) архитектуры. Итерации *конструирования* представляют базовый продукт, но еще требуется проделать работу, чтобы подготовить продукт к выпуску. Целью итераций *перехода* является подготовка приложения к выпуску (к отправке заказчику).

С учетом приведенной выше классификации итераций, USDP может быть представлен матрицей (рис. 1.11 и рис. 1.12). Как и большинство процессов объектно-ориентированного анализа и проектирования, фазы водопадного процесса, представленные Якобсоном, также включают в себя фазу анализа. На рис. 1.11 показано, где эта фаза совпадает с классическими фазами водопадного процесса. *Анализ* состоит из той части процесса анализа требований, в которой выбирают-

ся и соотносятся между собой базовые классы приложения. Кроме того, в USDP отсутствует своя фаза *интеграции*, обычно представленная в классическом водопадном процессе. Это произошло в связи с убежденностью Буча [15] в том, что объектно-ориентированные приложения могут и должны использовать непрерывную интеграцию. Другими словами, сразу после добавления новых частей исходное приложение интегрируется. Таким образом, отпадает необходимость в специальной фазе интеграции.

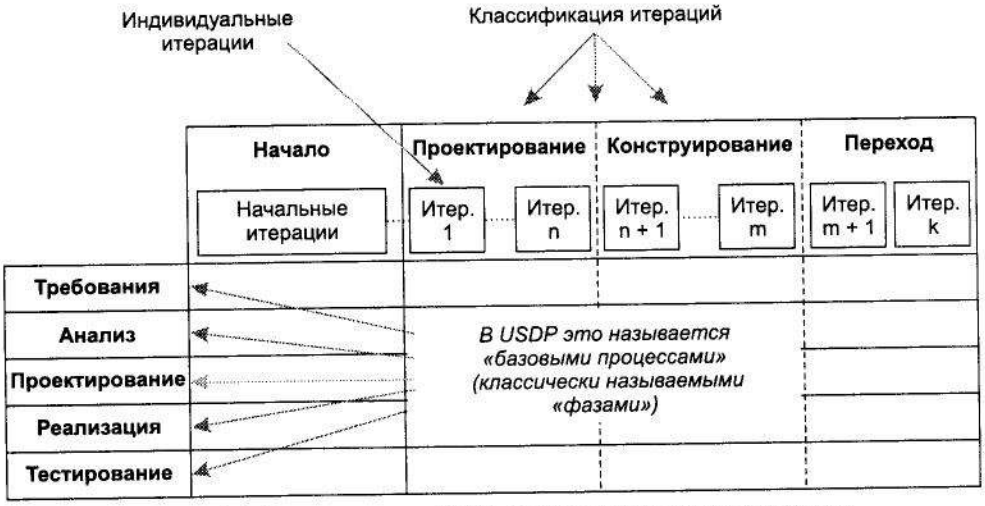


Рис. 1.11. Различия между USDP и стандартной терминологией (1)

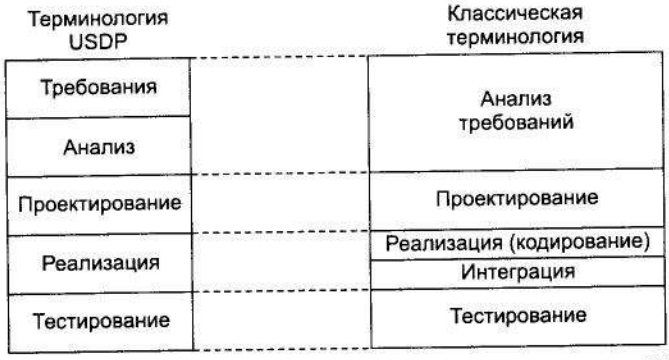


Рис. 1.12. Различия между USDP и стандартной терминологией (2)

Приблизительный тип и уровень работ для каждого базового рабочего итерационного процесса USDP показан на рис. 1.13. Большинство итераций в разной степени затрагивают практически все фазы водопадного процесса (базовые рабочие процессы). Начальные итерации содержат в основном анализ требований, непосредственно анализ, а также могут включать проектирование и реализацию для создания предварительного прототипа, который может быть использован для обсуждения проекта с заинтересованными сторонами. Итерации проектиро-

вания затрагивают в основном анализ требований, а также некоторую часть проектирования и реализации. Итерации конструирования включают в себя проектирование и реализацию, а итерации перехода — реализацию и тестирование.

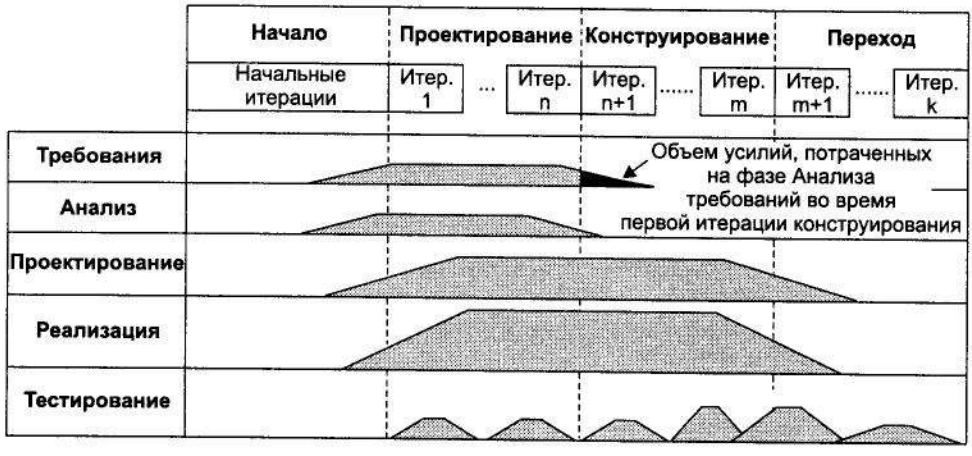


Рис. 1.13. Матрица USDP

USDP описывает шесть моделей (типов рассмотрения свойств приложения) (рис. 1.14). Эти модели будут обсуждаться в дальнейшем, при этом не всегда будет использоваться терминология USDP. Модель *вариантов использования* описывает случаи, в которых приложение будет использоваться. *Аналитическая* модель описывает базовые классы для приложения. Модель *проектирования* описывает связи и отношения между классами и выделенными объектами. Модель *развертывания* описывает распределение программного обеспечения по компьютерам. Модель *реализации* описывает внутреннюю организацию программного кода. Модель *тестирования* состоит из тестирующих компонентов, тестовых процедур и различных вариантов тестирования.

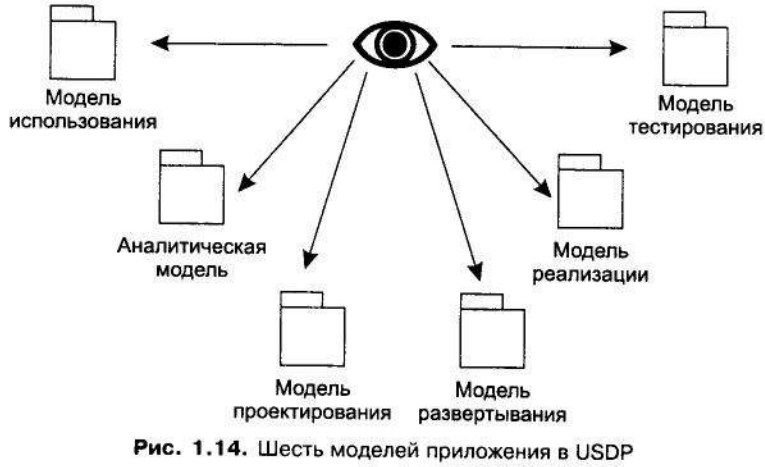


Рис. 1.14. Шесть моделей приложения в USDP

1.4.5. Сравнение процессов разработки

Характеристики особенностей, данные автором, для чистого водопадного, спирального и инкрементального процессов, приведены в табл. 1.1. Для каждого проекта конкретные факторы могут сильно влиять на представленные здесь оценки. Например, автор оценивает «Сбор метрических данных внутри проекта» как более тяжелый для чистого водопадного процесса, так как метрические данные, собранные о каждой фазе, не могут быть прямо использованы в рамках этого проекта (однако могут быть использованы для будущих проектов). В то же время в спиральном процессе данные, собранные на предыдущих итерациях (например, количество строк кода, введенных за час), могут использоваться на последующих итерациях.

Таблица 1.1. Издержки процессов разработки

Фактор	Чистый водопадный процесс	Итеративные процессы	
		Спиральный	Инкрементальный
Легкость контроля документации	Легче	Тяжелее	Тяжелее/Средне (пояснение 1)
Возможность взаимодействия с заказчиком	Тяжелее	Легче	Легче
Поддержание хорошего проектирования	Средне/Легче	Легче (пояснение 2)	Тяжелее
Сбор метрических данных, собранных в ходе проекта	Тяжелее	Средне/Легче	Средне/Легче

Далее следуют пояснения к табл. 1.1.

1. Инкрементальный процесс осуществим, если документация изначально полна и непротиворечива. Если документация полна и непротиворечива, то относительно небольшие шаги разработки достаточно легко документируются. При этом команда разработчиков получает прекрасную возможность попрактиковаться в обновлении документации, так как процесс повторяется много раз.
2. Шаги спиральной разработки достаточно немногочисленны, что позволяет проектировать на весьма высоком уровне, но в то же время их достаточно много, чтобы обеспечить проектировщикам растущее понимание проблем проекта. Это преимущество объясняет широкое использование спирального метода.

Для того чтобы принять решение, надо определить доступное время и сроки отсылок, требуемые инструктором. Обычно команды считают проведение двух итераций вполне достаточным. Проведение трех итераций дает некоторые преимущества, в том числе возможность собрать метрические данные в течение одной итерации (например, подсчитать количество строк кода), а затем использовать их на следующей итерации. Однако успеть совершить три итерации за один семестр может только очень целеустремленная, слаженная и деятельная команда.

ОДИН ИЗ СПОСОБОВ ВЫБОРА ПРОЦЕССА РАЗРАБОТКИ

Последовательность шагов, позволяющая команде студентов решить, каким процессом воспользоваться, может быть следующей.

1. Выберите, какой из процессов вам больше подходит: водопадный, спиральный или инкрементальный.
   - Для семестрового проекта обычно используется спиральный процесс.
   - Допускается комбинирование. Например, начинаем проект со спирального процесса, а под конец используем инкрементальный.
2. Решите вопрос о количестве итераций:
   - обычно для семестрового проекта достаточно двух итераций (в заключение каждой итерации необходимо координировать большое число артефактов);
   - три итерации предоставляют больше практики, однако это риск. Делайте первое приращение как можно меньшим;
   - три итерации способствуют накапливанию и использованию метрических данных. Используйте данные, полученные на предыдущих итерациях, для последующих итераций.
3. Разрабатывайте еженедельный график работ. Согласовывайте его с текущими заданиями. (Составление графиков работ обсуждается в следующей главе.)

В любом случае то, как много времени обычно требуется на согласование всех составляющих первой итерации, становится сюрпризом для участников проекта. По этой причине цели первой итерации должны быть предельно скромными. Обычно целью этой итерации становится получение простого графического интерфейса пользователя, так как он обычно прост для разработки и может быть продемонстрирован заказчику (подробнее — в главе 2). Первая итерация приучает участников к координации своих усилий. Это позволит команде на следующей итерации сконцентрироваться на функциональности и проектировании, которым посвящен материал глав 3–9. Если рабочая группа планирует третью итерацию, то ее целью, вероятнее всего, становится расширение функциональности и «причесывание» продукта.

Инкрементальная разработка может быть запланирована на самое окончание проекта, несмотря на то, что это может оказаться несколько амбициозным для семестрового проекта. К тому времени члены команды должны быть достаточно опытны, легко общаться между собой и не сомневаться в том, что их процесс протекает гладко.

1.5. Документация

1.5.1. Введение в документирование

Разработка программного обеспечения живет документацией. Документация может быть отделена от программного кода, а может быть тесно с ним связана. Для того чтобы понять важность исчерпывающей документации, представьте, что вам предложили участие в уже полным ходом идущем программном проекте. По уровню сложности это примерно соответствовало бы тому, как если бы вас заставили изучить, например, некоторые аспекты патологии крови. Чтобы сделать это, вам, скорее всего, захотелось бы получить общее представление о предмете,

узнать причины необходимости его изучения, затем от обзора постепенно перейти к изучению деталей, диаграмм, демонстрирующих внутренние связи, описаниям частных случаев и т. д. Упущения или противоречия могут привести к частичному, а возможно, и полному непониманию изучаемого материала.

Чтобы проиллюстрировать важность документирования программного обеспечения на различных уровнях, обратимся к фрагменту программного кода (листинг 1.1). Без полной документации этот код невозможно интерпретировать, поэтому его ценность невелика, если он вообще хоть чего-то стоит. Если мы добавим комментарии и сделаем имена функций более информативными (листинг 1.2), то получим несколько лучший результат. Этот результат даже может ввести нас в заблуждение, заставив поверить, что мы знаем значение и контекст кода; при этом последствия могут оказаться более катастрофичными, чем в том случае, если бы мы признали непонимание значения текста программы.

Листинг 1.1. Недокументированный код

```
int a(int i, char c)
{
        if(c== "m")
            if(i< 1000)
                return 0;
            else
                if(i< 10000)
                    return 500;
                else
                    return 1200;
        else
                return 1300;
}
```

Листинг 1.2. Частично документированный код

```
int tax(int anEarning, char aStatus)
{
        if(aStatus == "m")
            if(anEarning < 1000)
                return 0; // Женатые не облагаются налогом, <$1000
            else
                if(anEarning < 10000)
                    return 500;  // Женат, $1000-$10000
                else
                    return 1200; // Женат, >=$10000
        // Если не женат, то использовать ставку налога в размере $1300
        else
                return 1300;
}
```

Когда мы смотрим на тщательно документированный программный код (листинг 1.3), его значение становится намного понятнее. Благодаря документированию

мы узнаем очень важный новый факт, заключающийся в том, что приведенные ставки заработной платы действительны только для ограниченного отрезка времени. Ссылка на одно из требований в SRS является еще одной важной частью документации. Символы, которым предшествует @, относятся к зарезервированным словам Javadoc. Они будут подробно рассмотрены в главе 8.

Листинг 1.3. Документированный код

```
/**
* Этот метод реализует требование 4.3:
* "Установить действие налога с 01.09.98 по 31.12.99".
* @ author Э. Брауде
* @ version 2.3.4 (08.06.98)
* @ param anEarning: зарплата
* @ param aStatus: 'm' означает "женат", иное означает "не женат"
*/
int tax(int anEarning, char aStatus)
{ . . .
```

Однако история этой части программного кода еще не окончена. Благодаря управлению конфигурациями из документа *План управления конфигурациями ПО* мы узнаем, что текущей версией `tax()` является 2.7.3, а используется версия 2.3.4. Другими словами, фрагмент кода, рассмотренный нами, во многом уже не относится к делу, так как текущая версия 2.7.3 может выглядеть совсем по-другому.

До сих пор у нас еще не сложилось представление об общем положении вещей. Например, мы не знаем, работает ли версия 2.3.4 функции `tax()` как требуется? Какому классу принадлежит эта функция? Какому пакету? Даже если мы знаем имя пакета, то каково его предназначение? Как он соотносится с другими пакетами? То есть, документация обретает смысл не только из текста своего содержания, но еще и из контекста. «Лакмусовой бумажкой» для определения хорошего ведения документации является готовность нового инженера включиться в проект за разумное время. Подводя итоги, отметим, что проект — это полное множество согласованных, хорошо разработанных артефактов, включающее набор документов, результаты тестов и программный код.

1.5.2. Стандарты документации

В прошлом учреждение стандартов приносило огромные выгоды, получаемые от новых отраслей инженерии. Стандарты обеспечивают совместимость между проектами. Это означает, идеи или артефакты, разработанные для одного случая, могут быть перенесены и на другой. Стандарты улучшают понимание среди инженеров. Например, учреждение стандартов в электротехнике привело к большому прорыву вперед в способности производить широко используемые продукты и услуги. Наиболее крупные компании создали стандарты разработки программного обеспечения. Некоторые заказчики, такие как Министерство обороны США, настаивают, чтобы подрядчики следовали их стандартам.

Многие компании осознали, что просто издание и распространение стандартов не приводит к их принятию. Вы легко найдете толстые тома руководств по стандартам компании, собирающие пыль на книжных полках инженеров. Автор наблюдал также практически неиспользуемые карточки кратких ссылок на стандарты компании, что доказывает, что размер не является здесь решающим фактором. Даже принудительное посещение тренингов по стандартизации не склоняет инженеров следовать стандартам. Для того чтобы быть эффективными, стандарты должны восприниматься инженерами как нечто полезное для них, а не как набор препятствий. Кроме того, четкие и измеримые цели, требующие дисциплинированного и документированного подхода, обычно являются хорошим мотивом для разработчиков.

Хэмфри [51] предлагает командам коллективно решать, какой из стандартов ведения документации им применять. По-моему, такой выбор предпочтительнее, так как команда будет иметь мотив следовать своему собственному решению. Другим преимуществом является то, что компания при этом перебирает различные подходы, а в результате этого процесса, очень вероятно, могут быть приняты наиболее выгодные варианты на долгое время работы. Недостатком самостоятельного выбора стандарта командами является то, что группы, работающие в одной компании, зачастую выбирают разные стандарты. Это уменьшает возможности сравнения проектов и требует от инженеров, переключающихся на другой проект, изучения нового стандарта документации.

Организации должны делать стандарты как минимум простыми и понятными. Организации могут допускать некоторую гибкость и автономность в этом вопросе, но при этом рассчитывать на получение определенной стандартной информации для улучшения всего процесса производства. Например, организация вправе рассчитывать на получение данных, включающих в себя время, потраченное на разработку приложения, и объем программного кода, измеренные указанным способом. Такие стандартные измерения позволяют использовать данные по всей организации. Улучшение процесса включает в себя эволюционный метапроцесс (процесс, имеющий дело с другими процессами) внутри организации. Одним из примеров является модель зрелости возможностей (CMM), которая классифицирует организации, занимающиеся разработкой программного обеспечения, по пяти категориям возрастающих возможностей. Эта модель рассматривается в разделе 1.8.3.

Перечисленные ниже организации публикуют важные стандарты. Не все стандарты могут быть абсолютно актуальны, так как совещания, требуемые для их создания, проходят намного медленнее, чем появление новых технологий на рынке.

- Институт инженеров по электротехнике и радиоэлектронике (IEEE, www.ieee.org) в течение многих лет остается очень активен в создании стандартов документации программного обеспечения. Большинство стандартов разработаны различными комитетами, состоящими из опытных и ответственных инженеров-профессионалов. Некоторые из стандартов IEEE стали также стандартами ANSI. Эта глава, например, ссылается на три стандарта IEEE (разделы 1.6.5, 1.7.2 и 1.7.3).

- Международная организация по стандартизации (ISO) имеет огромное влияние во всем мире, особенно среди организаций производителей, имеющих дело с Евросоюзом (ЕС). ЕС предписывает следование стандартам ISO любой компании, имеющей дело со странами — членами Евросоюза, что является мощным стимулом для поддержания этих стандартов странами всего мира.
- Институт технологий разработки программного обеспечения (SEI) был учрежден Министерством обороны США в университете Карнеги-Меллон для поднятия уровня технологии программного обеспечения у подрядчиков Министерства обороны. Работа SEI также была принята многими коммерческими компаниями, которые считают улучшение процесса разработки программного обеспечения своей стратегической корпоративной задачей. В этой главе приводится важный стандарт, разработанный SEI, который называется Моделью зрелости возможностей (CMM).
- Консорциум по технологии манипулирования объектами (OMG, www.omg.org) является некоммерческой организацией, в которую в качестве членов входят около 700 компаний. OMG устанавливает стандарты для распределенных объектно-ориентированных вычислений. В частности, OMG использует унифицированный язык моделирования UML в качестве своего стандарта для описания проектов. Например, нотация интерфейса, использованная в описании технологии COM на рис. 1.4, является UML-нотацией.

Документы, сопровождающие проект, сильно различаются среди организаций, но примерно соответствуют водопадным фазам. Стандарт ISO 12207 является одним из примеров такого набора документов.

В этой книге применяется согласованное множество стандартов ведения документации, определенных IEEE. Несмотря на то что стандарты IEEE иногда требуют некоторой модернизации, они помогают инженерам справляться с большинством проблем, возникающих с документацией, что позволяет им максимально сконцентрироваться на разработке. Когда стандарты ведения документации не применяются, инженерам приходится затрачивать огромное количество времени, самостоятельно приводя документы в порядок, что равносильно очередному изобретению колеса. Типичный набор документации с использованием терминологии IEEE показан на рис. 1.15. В примере в конце этой главы используются большинство документов из этого набора. Использование набора документации для водопадного процесса не означает, что обязательно должна использоваться водопадная модель. Однако если она не используется, то необходимо провести обновление всех документов и пополнять их каждый раз, когда применяются водопадные фазы. Это означает, что документы должны быть очень хорошо организованы. Это чем-то напоминает устройство библиотеки, в том смысле, что только в хорошо организованной библиотеке очень легко искать, брать и возвращать книги.

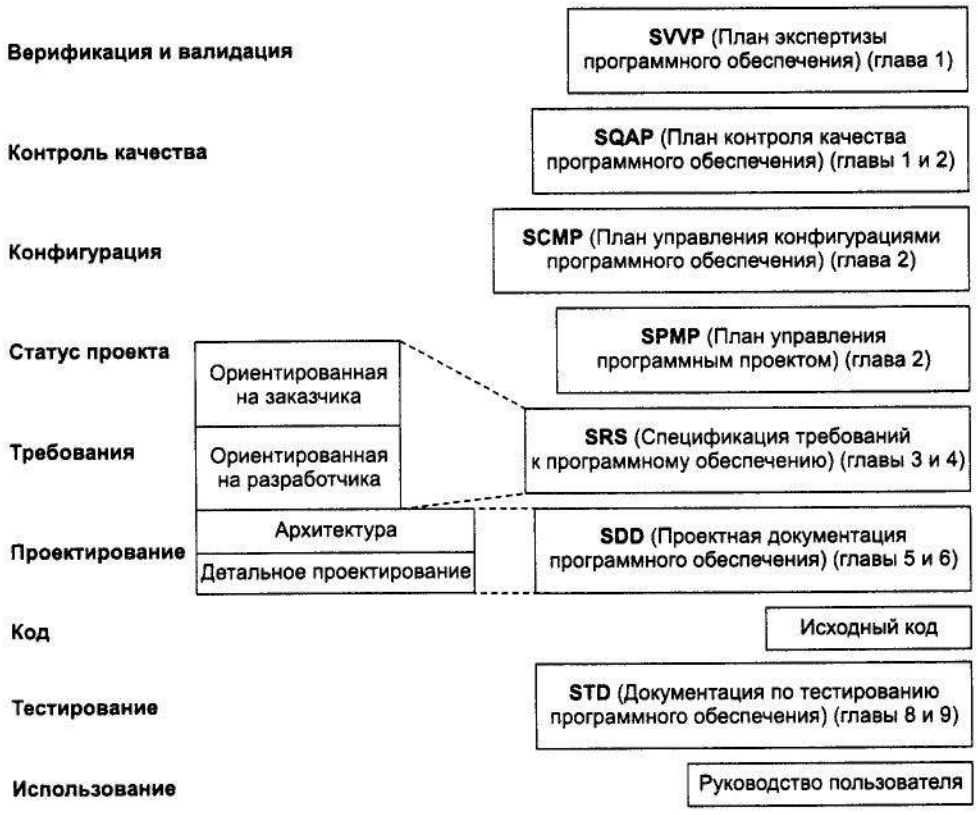


Рис. 1.15. Документация проекта

Ниже приводится описание каждого документа из набора IEEE с ссылками на полные описания в этой книге. Другие стандарты внутренне организованы по тому же принципу.

- SVVP (Software Verification and Validation Plan): План экспертизы программного обеспечения. Этот план определяет, каким образом и в какой последовательности должны проверяться стадии проекта, а также сам продукт на соответствие поставленным требованиям. Верификация — это процесс проверки правильности сборки приложения; валидация проверяет тот факт, что собран требуемый продукт. Полностью это описано в разделе 1.6.6. Зачастую валидацию и верификацию осуществляют сторонние организации, в этом случае экспертиза называется независимой (IV&V — Independent V&V).
- SQAP (Software Quality Assurance Plan): План контроля качества программного обеспечения. Этот план определяет, каким образом проект должен достигнуть соответствия установленному уровню качества. Подробные объяснения можно найти в разделе 1.6.5, в примере в конце этой главы и в главе 2.

- SCMP (Software Configuration Management Plan): План управления конфигурациями программного обеспечения. SCMP определяет, как и где должны храниться документы, программный код и их версии, а также устанавливает их взаимное соответствие. Было бы крайне неразумным начинать работу без такого плана, так как самый первый созданный документ обречен на изменения, а мы должны знать, как управлять этими изменениями до того, как мы начнем составлять документ. В этой главе приводится описание IEEE SCMP. Для примера мы ограничимся довольно простым планом, приведенным в конце этой главы. Средние и большие компании, как правило, стараются выработать единое управление конфигурациями для всех своих проектов. Таким образом, инженерам требуется только научиться следовать предписанным процедурам в соответствии с SCMP. SCMP описывается далее в разделе 1.7 и в примере в конце главы.
- SPMP (Software Project Management Plan): План управления программным проектом. Этот план определяет, каким образом управлять проектом. Обычно он соответствует известному процессу разработки, например стандартному процессу компании. SPMP обсуждается в главе 2, которая посвящена управлению проектом.
- SRS (Software Requirements Specification): Спецификация требований к программному обеспечению. Этот документ определяет требования к приложению и является подобием контракта и путеводной нити для заказчика и разработчиков. SRS рассматривается подробнее в главах 3 и 4.
- SDD (Software Design Document): Проектная документация программного обеспечения. SDD представляет архитектуру и детали проектирования приложения, обычно с использованием диаграмм объектных моделей и потоков данных. Подробнее о SDD см. в главах 5 и 6.
- STD (Software Test Documentation): Документация по тестированию программного обеспечения. Этот документ описывает, каким образом должно проводиться тестирование приложения и его компонентов. Подробные пояснения см. в главах 8 и 9.

Иногда в проектах привлекается дополнительная документация (например, см. [56]). Документация для итеративной разработки может быть организована двумя способами. Некоторые документы, в частности SDD, могут содержать свою версию для каждой итерации. Другой способ — дописывать дополнения, которые появляются по мере развития приложения.

Унифицированный язык моделирования (UML) был разработан для стандартизации описания программных проектов, в особенности объектно-ориентированных. UML был принят в качестве стандарта консорциумом OMG. Эта книга использует UML в качестве нотации для различных артефактов, включая проектирование и физическую конфигурацию исходных файлов. Выборочно UML описан непосредственно в тексте книги.

ОДИН ИЗ СПОСОБОВ ОПРЕДЕЛЕНИЯ НЕОБХОДИМОСТИ ТОЙ ИЛИ ИНОЙ ДОКУМЕНТАЦИИ

Итоговый перечень документации, необходимой для ведения любого проекта, независимо от того, используются ли в нем признанные стандарты документации, такие как IEEE, можно определить следующим образом (символом «*» отмечены документы стандарта IEEE, которые могут быть полезны для организации документации).

1. Зафиксируйте (выявите) документы и программный код, которые должны быть доступны, во избежание хаоса.
   - SCMP* — раздел 1.7.3.2.
2. Установите, кто что будет делать и когда они будут это делать.
   - SPMP* — глава 2.
3. Задокументируйте, что должно быть реализовано.
   - Для себя, для заказчика, для вашей команды;
   - SRS* — главы 3 и 4.
4. Задокументируйте проектирование приложения до того, как приступать к программированию.
   - SDD* — главы 5 и 6.
5. Напишите и задокументируйте программный код.
   - Основы кодирования рассматриваются в главе 7.
6. Задокументируйте проводимые тесты.
   - Благодаря этому тесты могут быть воспроизведены и дополнены.
   - STD — главы 8 и 9.

Детали

Эту часть можно пропустить при первом чтении и вернуться к ней после прочтения последующих глав. Однако в любом случае владеть излагаемым здесь материалом необходимо, поскольку он касается вопросов производства качественного программного обеспечения.

1.6. Качество

Имеется большая разница между тем, чтобы просто запрограммировать некоторую функцию и тем, чтобы изготовить ее как качественный продукт. В первом случае мы имеем код, о котором нельзя сказать ничего определенного, кроме того, что он компилируется и, по всей видимости, «работает». Во втором случае мы получаем код, который:

- удовлетворяет ясно сформулированным требованиям;
- проверяет входные данные и предсказуемо реагирует на некорректные входные данные;
- тщательно проинспектирован не только самим автором кода, но и другими разработчиками;

- прошел исчерпывающее многостороннее тестирование;
- тщательно документирован;
- имеет надежную оценку степени дефектности (процента ошибок).

Разница здесь примерно того же порядка, что и между изготовлением самодельной полочки для книг и изготовлением балки моста, спроектированной с учетом возможных нагрузок транспортного и пешеходного движения.

Аналогично, высококачественный программный продукт обычно является:

- расширяемым (готовым к возможным изменениям для расширения функциональности);
- развиваемым (легко адаптируемым к изменению требований);
- переносимым (пригодным к использованию на нескольких платформах);
- общим (применимым к нескольким различным ситуациям).

Нашей целью является разработка стандартов для данных характеристик и создание продуктов, которые удовлетворяли бы этим специфицированным стандартам. Для этого мы должны научиться измерять качество, численно задавать требуемые уровни качества и уметь контролировать процесс достижения заданного уровня качества.

1.6.1. Метрики

Использование метрик, то есть количественных характеристик, типично для различных инженерных дисциплин. Например, транспортные потоки характеризуются количеством автомобилей в час, а в механике говорят о предельных нагрузках. В проектировании программ также используются различные количественные характеристики, такие как число строк кода, число классов, количество дефектов, выявленных за месяц, число функций в классе.

Трактовка метрик определяется контекстом, в котором эти метрики используются. Например, если мы имеем две программы, обладающие одинаковой функциональностью, одинаково эффективные, одинаково надежные и хорошо структурированные, то более предпочтительной, по-видимому, следует считать ту из них, которая имеет меньшее число строк кода. В другом же контексте большее число строк кода может означать большую производительность. При наличии достаточной статистической базы метрика *число строк кода* может быть очень полезна, особенно в сочетании с другими характеристиками. Например, если считать, что метрика надежности приложения остается на требуемом уровне, а увеличение количества строк кода отражает расширение возможностей приложения, становится понятным, что чем больше строк кода будет написано за час, тем лучше. Обычно ведется учет нескольких различных метрик.

В этой книге мы рассмотрим различные метрики и их использование. Метрики, которые необходимо знать практически всегда, включают в себя:

- объем выполненной работы, измеренный в физических единицах (например, число строк кода);

- время, затраченное на выполнение работы;
- степень дефектности (число дефектов на 1000 строк кода, число дефектов на страницу документации и т. д.).

Кроме того, следует применять также относительную оценку качества работы по шкале от 0 до 10.

Предварительные или желаемые значения метрик заранее прогнозируются, а затем сравниваются с полученными результатами. Например, наша организация ожидает 0,2 дефекта на страницу описания требований (в среднем — один на пять страниц, как известно по прошлым проектам). Нашей целью для данного проекта может являться 0,15 дефекта на страницу. Реальная же степень дефектности может составлять 0,17. Это говорит о том, что наши методы лучше использованных в прошлом, однако недостаточно хороши для достижения поставленной цели.

1.6.2. Процесс контроля качества

Ответственность за качество артефакта в первую очередь несет человек, создающий этот артефакт. Но, как бы то ни было, «один в поле не воин». Всем нам требуется сторонний взгляд на выполняемую нами работу (в том числе и автору этой книги!). Этот взгляд необходим, чтобы избежать недальновидности, нереалистичной самооценки и застоя. Процесс контроля качества является также общественной обязанностью. Каждая часть работы, выполненная инженером, должна быть детально проверена по крайней мере одним человеком, желательно независимым от автора работы.

В дополнение к ответственности индивидуальных разработчиков и проверкам работы их коллег многие организации определили процесс раздельной систематической и полной проверки, в дальнейшем — *контроль качества* (QA — quality assurance). В функции контроля качества входят проверки, инспектирование (формальный тип проверки, приводимый ниже) и тестирование. Контроль качества должен начинаться вместе с запуском каждого проекта (рис. 1.16). Лучше всего привлекать контроль качества и для проверки правильности используемого процесса и актуальности документации. Представитель группы контроля качества часто принимает участие в инспектировании. В идеале контроль качества должен осуществляться некоторой независимой организацией. Многие компании слишком малы для осуществления столь сложного контроля качества, и в этом случае сами инженеры осуществляют функции контроля качества по отношению к работе друг друга.

1.6.3. Методы «белого ящика» и «черного ящика»

В случае контроля качества методом *«черного ящика»* приложение (или какая-либо его законченная часть) анализируется как целое. Этот метод используется для проверки того, что приложение (его часть) отвечает предъявляемым требова-

ниям. Контроль качества методом *«белого (стеклянного) ящика»* осуществляется на уровне компонентов, из которых построено тестируемое приложение (его часть). Можно провести такую аналогию: если вы проверяете работу телевизора, просто включая и выключая его и переключая каналы, то вы применяете метод «черного ящика»; если же вы тестируете телевизор, анализируя его работу на уровне взаимодействия микросхем, то есть компонентов, из которых телевизор собран, вы применяете метод «белого ящика». Промежуточным является метод *«серого ящика»* (ситуация, когда вы проверяете основные компоненты телевизора). Заметим, что грань между «белым» и «серым» здесь зачастую расплывчата.

Рис. 1.16. Осуществление контроля качества

Чаще всего о методах «черного» и «белого ящика» говорят в контексте тестирования. Однако эти методы применимы и к другим способам контроля качества. Метод «белого ящика» основан на рассмотрении анализируемого артефакта с точки зрения его структуры, формы и назначения; здесь применяются формальные методы и рассматриваемое в следующем разделе инспектирование. Метод «черного ящика» задается вопросом: «Обладает ли построенный объект требуемым поведением?». Подробнее эти методы изучаются в главах 8 и 9.

1.6.4. Введение в инспектирование

Инспектирование — это техника «белого ящика» для обеспечения качества. Инспектирование состоит в проверке частей проекта (требований, результатов проектирования, программного кода и т. п.) на наличие дефектов. Инспектирование проводит группа коллег автора работы. Мы предлагаем начинать инспектирование очень рано, так как оно должно начинаться вместе с появлением первой документации по проекту. Поскольку инспектирование изначально было предложено для улучшения качества программного кода, то его часто называют *инспектированием кода*, однако было показано, что инспектирование достигает наилучших

результатов, если начинается как можно раньше, еще задолго до появления текста программ.

Основная идея инспектирования хорошо описана Фагином [30], который заметил, что автор в большинстве случаев способен исправить дефект своей работы, когда тот обнаружен. Таким образом, инспектирование необходимо использовать хотя бы для того, чтобы автор замечал дефекты в своей работе до того, как представит ее начальству. Подразумевается, что инспектирование выполняется коллегами по разработке. Это можно представить в виде следующего заключения:

- Авторы обычно в состоянии исправить найденные дефекты.
  - **Следствие:** помогайте авторам находить дефекты до завершения работы.
  - **Следствие:** пусть их коллеги ищут дефекты.

Принцип инспектирования может быть обобщен четырьмя правилами.

1. *Вскрытие дефектов.* Из инспектирования намеренно исключается исправление дефектов. Процесс исправления предоставлен автору. Во время инспектирования ни минуты времени не должно быть потрачено на обсуждение способов исправления дефекта. Все обсуждения должны проходить после окончания инспектирования.
2. *Участие коллег.* Инспектирование проводится внутри группы разработчиков программного обеспечения и не предполагает вовлечения в отношения «начальник—подчиненный». Инспектируется текущая работа, а не способности ее автора. Автор несет ответственность только за конечный продукт, тогда как инспектирование проводится до того, как работа сдана. Однако работа, представленная автором для инспектирования, должна быть лучшим вариантом, но ни в коем случае не черновиком. Было бы пустой тратой ресурсов группы искать, находить и описывать дефекты, которые автор, приложив некоторые усилия, в состоянии найти сам.
3. *Распределение ролей.* Каждый из участников проекта берет на себя одну из следующих ролей. При нехватке кадров один человек может выполнять сразу две роли. Обычно ведущий может одновременно быть и корректором. Однако для достижения беспристрастной проверки автор не должен выполнять никаких других ролей.
   - *Ведущий* ответственен за правильное проведение инспектирования. Ведущий также является инспектирующим.
   - *Автор* несет ответственность за свою работу и за исправление всех найденных в ней дефектов. Автор является одновременно и инспектирующим, проводящим время в поиске новых дефектов.
   - *Корректор* отвечает за деятельность команды и направляет ее в нужное русло. Корректор принимает участие в инспектировании.
   - *Регистратор* отвечает за учет описания и классификацию дефектов, как это принято в команде. Регистратор также участвует в инспектировании.

Далее приводятся дополнительные роли инспектирующих. Необходимость их введения зависит от специфики инспектируемых артефактов.

- *Профильный инспектирующий* проверяет артефакт по одному конкретному критерию (например, на надежность).
- *Специализированный инспектирующий* — это специалист в той области, которой принадлежит инспектируемый артефакт (например, эксперт по радиолокации для приложения управления радиолокатором.)
- *Простой инспектирующий* не имеет специальной роли, кроме проверки материала на наличие дефектов.

4. *Тщательная подготовка.* Участникам инспектирования необходимо подготовиться к нему так же детально, как и самому автору. Инспектирование не является просто обзором, обсуждением руководства или образовательным семинаром. Инспектирующие должны работать на том же уровне детализации, что и автор. (Как раз это и делает инспектирование таким дорогостоящим.)

В процессе разработки инспектирование должно быть начато как можно раньше. Например, сразу же должны подвергнуться инспектированию требования к приложению. Аналогично, могут быть проверены планы управления проектом и конфигурациями.

Для проведения инспектирования требуется выполнение следующих шагов (рис. 1.17).

1. Процесс инспектирования начинается с *планирования*. Планирование включает в себя выбор метрик, по которым будет проводиться инспектирование, а также выбор инструментов для сбора и анализа полученных данных.
2. В ходе проекта ответственные лица просто должны назначать, какая группа сотрудников участвует в инспектировании и какие части проделанной работы должны быть проверены. Части должны быть логически завершенными. Итог инспектирования подводится на 1–4-часовом собрании (обсуждаемом ниже в пункте 5).
3. При необходимости может быть организован *обзорный семинар* для лучшего понимания объекта инспектирования. Однако такой способ обсуждений является весьма дорогостоящим, и его не следует использовать без крайней необходимости.
4. Следующая фаза состоит в *подготовке*. Инспектирующие проверяют работу в полном объеме на своих рабочих местах (например, проверяют, соответствует ли инспектируемый программный код детальному проекту). Еще одним достоинством является то, что в этом процессе задействованы разные люди. Все это делает процесс инспектирования очень ценным, однако и достаточно дорогостоящим. Инспектирование не является простым просматриванием материала, так как инспектирующий работает на том же уровне детализации, что и автор. Инспектирующие обычно заносят все дефекты в базу данных (например, доступную через сеть) вместе с описаниями и классификацией. Это помогает избежать дублирования и мини-

мизирует время на собрания. Некоторые предпочитают фиксировать дефекты на бумаге, другие считают информативной метрикой количество инспектирующих, обнаруживших данный дефект.

5. Как только каждый из участников процесса готов, проводится *инспекционное собрание*, в ходе которого участники выполняют свои роли.
6. Как правило, автор в состоянии исправить все дефекты. Это называется фазой *доработки*. Если инспекционное собрание решает, что дефекты настолько серьезны, что требуется повторное инспектирование данного объекта, то объект еще раз запускается в процесс.
7. Если дефекты появляются из-за недопонимания или просто плохого представления, то, возможно, необходимо организовать встречу по *анализу причин* появления дефектов. Опять же в силу дороговизны такие встречи должны проводиться только в исключительных случаях.
8. Окончательное *собрание по завершению работы* должно быть коротким. На нем корректор и автор убеждаются в том, что все дефекты исправлены. Однако это не предполагает детальной ревизии всей работы корректором. Все исправления остаются на совести автора, ответственного за свою работу.
9. Как и после других процессов, группа встречается для обсуждения самого процесса инспектирования и решает, как он может быть улучшен. Сюда же входит и улучшение инспекционных списков контрольных вопросов.

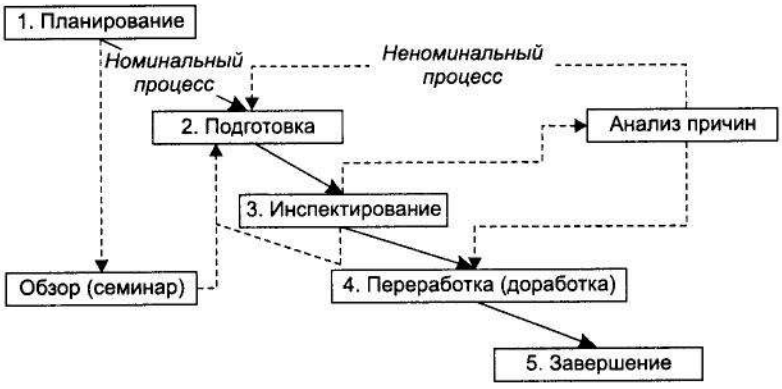


Рис. 1.17. Процесс инспектирования

Среднее относительное время для каждой фазы процесса инспектирования, используемое, например, одним из клиентов автора, приведено в табл. 1.2. Для определенности в качестве времени, отведенного на инспекционное собрание, взят один час. Самостоятельные компании и группы разработчиков ведут учет времени, потраченного на инспектирование, и объема проверенной работы с целью их дальнейшего использования при оценке будущих инспектирований. Время, приводимое в табл. 1.2, может обеспокоить непосвященных, сомневающихся в том, что проверка кода может отнимать столько времени. Производство профессиональных качественных продуктов действительно занимает значительное

количество времени. На обнаружение ошибок в конечном итоге затрачивается намного больше времени, чем предполагалось изначально. По оценке Джеани и Лалли [36], на инспектирование уходит порядка 10–15 % бюджета разработки.

Таблица 1.2. Стоимость 100 строк кода, не считая комментариев, в человеко-часах по оценкам одной компании

Планирование	1 человек × 1 час
Обзор: дополнительно	3–5 человек × 1 час
Подготовка	2–4 человека × 1 час
Инспекционное собрание	3–5 человек × 1 час
Доработка	1 человек × 1 час
Анализ: дополнительно	3–5 человек × 1 час
Всего: примерно	7–21 человеко-часов

В различных источниках (например, [30]) отмечается, что инспектирование, несмотря на большие затраты времени экспертов и дороговизну проведения, вполне окупает себя. Если начать подсчитывать стоимость инспектирования, то можно схватиться за голову, однако всегда следует сравнивать эту стоимость со стоимостью пропущенных дефектов. В конечном итоге, все дефекты должны быть найдены и исправлены. Если это не делается вскорости после того, как дефект был внесен, то стоимость исправления катастрофически возрастает. Сравнение стоимости обнаружения и исправления дефекта во время инспектирования и во время интеграции приводится в табл. 1.3. Эти оценки, полученные от клиента автора, являются весьма сдержанными.

Таблица 1.3. Оценка количества времени, затрачиваемого на один дефект

	Дефект, найденный во время инспектирования	**Дефект, найденный во время интеграции**
Количество часов на отыскание	0,7–2	0,2–10
Количество часов на исправление	0,3–1,2	9+
Всего:	1,0–3,2	9,2–19+

В данном примере (см. табл. 1.3) на отыскание каждого дефекта уходит от 0,7 до 2 часов, в случае если поиск ведется в рамках процесса инспектирования. С другой стороны, на это уходит от 0,2 до 10 часов при поиске дефектов в уже собранном приложении (то есть по окончании процесса разработки). Собрав эти данные для конкретной группы разработчиков и проекта, можно предсказать время, необходимое на тестирование, и т. д. Например, чтобы вычислить среднее время, затрачиваемое на поиск дефекта, необходимо количество часов, проведенных за тестированием, разделить на количество найденных дефектов.

Конкретные примеры выгоды от проведения инспектирования такими компаниями, как IBM, ICL и Standard Bank, приводятся в [37]. Основная выгода заключается в нахождении дефектов на ранних стадиях процесса разработки, и здесь цель оправдывает средства. По оценкам многих компаний, чем раньше начинают проводить инспектирование, тем больше экономия. В частности, непомерно вы-

сокий процент дефектов в продукте, как правило, является результатом искажения исходных требований. Например, представьте себе цену отыскания и исправления следующего дефекта в исходных требованиях.

- Реализованное требование: если температура достигает рубежа в 5,02 % до максимально допустимого предела, то, как определено стандартом 67 892, мотор должен быть отключен.
- Настоящее требование: если температура достигает рубежа в 5,02 % до максимально допустимого предела, то, как определено стандартом 67 892, работа мотора должна быть завершена.

Составитель требований посчитал, что «отключение» и «завершение работы» — эквивалентные понятия. Однако «завершение работы» для приложения может означать процесс поэтапного выключения элементов приложения в определенном порядке. «Отключение» может означать одновременное выключение всех элементов. Для точного определения значений таких требований необходимы дополнительные спецификации.

К сожалению, очень вероятно, что тестирование с достижением «максимально допустимого предела» может быть проведено только незадолго до завершения процесса разработки, и в ходе этого тестирования приложение будет просто неожиданно отключаться по непонятной причине. Этот дефект тривиален для специалистов по инспектированию *требований*, однако он может быть далеко не тривиальным для *системных тестеров*.

Как отмечается в примере по SQAP в конце этой главы, *обзоры*, вообще говоря, не являются инспектированием. Обзоры — это собрания, на которых обсуждается как уже завершенная, так и текущая работа. Примером тому является обзор, в котором может обсуждаться альтернативная архитектура приложения. Хотя обзоры являются неотъемлемой частью проекта, они не требуют такой детальной подготовки, как инспектирование. К тому же участникам обзоров, в отличие от ситуации с инспектированием, не надо играть никаких ролей.

ОДИН ИЗ СПОСОБОВ ПОДГОТОВКИ И ПРОВЕДЕНИЯ ИНСПЕКТИРОВАНИЯ

1. Включите инспектирование в график работы над проектом:
   - спланируйте инспектирование всех фаз начиная с подготовки требований;
   - выделите время на подготовку и собрание.
2. Подготовьтесь к накоплению данных по инспектированию:
   - учитывать количество дефектов на одну логически законченную часть работы (например, на тысячу строк кода), потраченное время;
   - разработать классификацию дефектов по описанию, степени серьезности и типу;
   - решить, кто, где и как должен хранить и использовать метрические данные;
   - при невыполнении: назначить одного ответственного за это человека;
   - обычно при устранении ошибок данные сбрасываются.
3. Обозначить роли участников инспектирования:
   - «третий — не лишний» (автор; посредник-регистратор; корректор);
   - «одна голова — хорошо, а две — лучше» (автор; инспектирующий);
4. Обеспечить подготовленность каждого участника.
5. Классифицировать ошибки перед обсуждением на инспекционном собрании.

Для выполнения этих шагов требуется значительная организационная работа. По личному опыту автора, главным испытанием для организаций оказывается систематизация и использование накопленных метрических данных. Это легко объяснимо недостатком простого планирования. Эта ситуация постепенно меняется к лучшему по мере того, как компании осознают стратегическое значение своих данных и как студенты, закончившие современные курсы по программным технологиям, оказываются готовыми к использованию метрик. Работа студенческой команды предоставляет прекрасную возможность попрактиковаться в сборе и в применении метрик. Самое простое решение задачи накопления и использования метрических данных — назначить ответственного за это человека. Данные в основном используются для измерения качества и оценки требований для будущих проектов.

Инспектированию подвергается каждая фаза каждой части проекта. Однако ввиду ограниченности ресурсов, доступных студенческим командам, возможно выборочное инспектирование частей проекта и конечного продукта. В главе 2 приводится описание системы «опеки», в которой каждый член команды опекается своим коллегой. Этот «опекун» должен участвовать в инспектировании работы своего подопечного. При жестких временных рамках инспектирование может проводиться в одиночку «опекуном». Это, конечно, не идеальный вариант, но все же лучше, чем полное отсутствие инспектирования.

1.6.5. План контроля качества (SQAP): стандарт IEEE

Для учета всех факторов контроля качества удобно пользоваться списками контрольных вопросов. Такие списки содержат пункты, которые необходимо последовательно проверить, они особенно удобны при использовании стандартов проектирования. Первые шесть разделов содержания стандарта IEEE 730 перечислены ниже. Остальные разделы будут рассмотрены в главе 2. В основном стандарт IEEE больше приспособлен для обширных проектов, но подходит и для малых. Стандарт напоминает нам обо всех факторах, которые должны быть учтены, и позволяет легко отбросить части, не относящиеся к конкретному проекту.

1. **Цель**
2. **Задействованные документы**
3. **Управление**
   - 3.1. Организация
   - 3.2. Задачи
   - 3.3. Ответственность
4. **Документация**
   - 4.1. Цель
   - 4.2. Минимальные требования к документации
   - 4.3. Прочее

5. Стандарты, практики, соглашения и метрики

5.1. Цель

5.2. Содержание

6. Обзоры и аудиты

6.1. Цель

6.2. Минимальные требования

6.2.1. Обзор требований к программному обеспечению

6.2.2. Предварительный обзор проектных решений

6.2.3. Критический обзор проектных решений

6.2.4. Обзор SVVP

6.2.5. Аудит функциональности

6.2.6. Аудит физических компонентов

6.2.7. Аудит процесса

6.2.8. Обзор управления

6.2.9. Обзор SCMP

6.2.10. Окончательный обзор

6.3. Прочее

7–15. См. следующую главу (раздел 2.12.2).

Данный документ определяет:

- кто будет нести ответственность за качество (раздел 3 этой главы) — физическое лицо, менеджер, группа, организация и т. п;
- какая документация требуется (раздел 4 этой главы) — (глава 2, Управление проектом);
- какие методы будут использоваться для гарантии качества (раздел 5 этой главы) — инспектирование, доказательство корректности, тестирование и т. д;
- какие мероприятия должны быть проведены в ходе управления процессом (раздел 6 этой главы) — собрания, аудиты, обзоры и т. п.

1.6.6. Верификация и валидация

Верификация и *валидация* (V&V — Verification and Validation) являются составной частью плана контроля качества. Верификация отвечает на вопрос «Правильно ли построен наш объект?». Или более детально: «Делаем ли мы на данной фазе в точности то, что было запланировано в предыдущей фазе?». Валидация же отвечает на вопрос: «Делаем ли мы то, что нужно?». Или другими словами: «Отвечает ли построенный объект пожеланиям и нуждам заказчика?». Разницу между этими понятиями иллюстрирует рис. 1.18. К сожалению, в литературе термины *верификация* и *валидация* часто путаются. Мы же далее будем использовать эти термины именно в соответствии с данным определением.

Верификация:
мы собираем предмет правильно?

Валидация:
мы собираем правильный предмет?

Рис. 1.18. Разница между валидацией и верификацией

Чтобы закрепить наше понимание различия между верификацией и валидацией, приведем простой пример. Заказчик хочет, чтобы мы разработали приложение, которое «решает линейные уравнения вида $ax + b = c$». Мы формулируем это пожелание в виде следующих детальных требований.

1. Пользователь вводит числа a, b и c, состоящие не более чем из десяти цифр, включая максимум четыре цифры после запятой.
2. Приложение отыскивает решение уравнения $ax + b = c$ с точностью до 1/1000.

Затем мы пишем программу, реализующую эти требования.

Валидация нашей программы состоит в прогоне набора тестов. Например, мы вводим $a = 1$, $b = 1$, $c = 1$ и проверяем, что программа выдает $x = 0$. (Подробнее о тестировании — в главах 8 и 9). Тестирование выявляет наличие каких-либо дефектов в программе, но не позволяет убедиться, что дефекты в программе отсутствуют. Иными словами, на основе валидации мы не можем гарантировать, что наша программа не содержит дефектов.

Верификация состоит в том, что мы анализируем процесс построения программы и убеждаемся, что в этом процессе все сделано правильно. Процесс разработки нашей программы включает в себя следующие шаги.

1. Получение требований заказчика.
2. Подготовка детальных требований.
3. Подготовка кода программы.
4. Тестирование программы.

Верификация будет заключаться в следующих проверках.

- $1 \rightarrow 2$. Соответствуют ли детальные требования тому, что заказчик действительно хочет? Здесь верификация может включать в себя следующие пункты: может ли быть $a = 0$? Если да, то что в этом случае должна делать программа? Если нет, то, возможно, заказчику требуется решать уравнения вида $x + b = c$; вообще, возможно, что заказчику естественнее задавать линейное уравнение в каком-либо другом виде; обсуждение с заказчиком

вопроса точности задания входных данных и получаемого решения и т. д. Инспектирование требований также относится к процессу верификации.

- 2 → 3. Реализует ли код программы сформулированные требования? Это подразумевает инспектирование кода, в процессе которого код анализируется с точки зрения соответствия каждому отдельному требованию. Анализ может включать в себя математические доказательства или основываться на них.
- 3 → 4. Достаточно ли выполненный (или подготовленный) набор тестов охватывает область применимости программы? Инспектирование тестов (главы 8 и 9).

1.6.6.1. Стандарт IEEE 1012-1986

Стандарт IEEE 1012-1986 для верификации и валидации задает основу, с помощью которой определяется, каким образом должны проводиться верификация и валидация. Стандарт 1012-1986 был переутвержден в 1992 году. В идеале одинаковые процедуры верификации и валидации должны применяться ко всем проектам компании. Лучше всего, если валидацией и верификацией занимается сторонняя организация (независимая экспертиза, обозначаемая IV&V). Оглавление этого стандарта приводится далее.

1. **Цель**
2. **Задействованные документы**
3. **Определения**
4. **Обзор V&V**
   - 4.1. Организация
   - 4.2. Основной график работ
   - 4.3. Ресурсы
   - 4.4. Ответственность
   - 4.5. Инструменты, техники, методики
5. **Жизненный цикл V&V**
   - 5.1. Управление V&V
   - 5.2. Фаза концепции V&V
   - 5.3. Фаза требований V&V
   - 5.4. Фаза проектирования V&V
   - 5.5. Фаза реализации V&V
   - 5.6. Фаза тестирования V&V
   - 5.7. Фаза установки и прекращения V&V
   - 5.8. Фаза работы и сопровождения V&V
6. **Отчетность по V&V программного обеспечения**
   - 6.1. Обязательные отчеты
   - 6.2. Дополнительные отчеты

7. **Административные процедуры V&V**
   - 7.1. Отчеты и решения относительно отклонений
   - 7.2. Политика итеративности задачи
   - 7.3. Политика уклончивости
   - 7.4. Стандарты, практики и соглашения

ОДИН ИЗ СПОСОБОВ ПРОИЗВОДСТВА КАЧЕСТВЕННОГО ПРОГРАММНОГО ПРОДУКТА

1. Определитесь с уровнем качества:
   - минимально: количество дефектов на единицу кода;
   - в группе: количество неверных требований; количество отсутствующих в проекте классов; количество дефектов в тестировании; количество дефектов, найденных в ходе работы приложения;
   - персонально: учесть количество дефектов в коде, компилировать, тестировать модули по отдельности.
2. Включите инспектирование и обзоры в график работы над проектом:
   - составление графика работ — в следующей главе;
   - следуйте схеме инспектирования (см. рис. 1.17).
3. Документируйте ваши требования по качеству и все действия:
   - используйте один из стандартов ведения документации;
   - SQAP (пример в конце главы); если позволяет время — SVVP.

Может оказаться, что первый шаг (задание определенного уровня качества) невозможно выполнить. В основном это касается групп разработчиков, не имеющих статистических данных по предыдущим проектам. В этом случае следует использовать известные и описанные в литературе характеристики. Каким вы выбрали максимальное количество дефектов на тысячу строк кода? Может ли быть хорошим стандартом «один дефект не более чем средней тяжести на тысячу строк кода»? Выберите такой процент дефектности, который, по вашему мнению, отражал бы ваш стандарт качества. Пожалуй, будет поучительным сравнение реального количества ошибок с оценками их числа, заложенными в проект. У людей есть тенденция забывать полученные результаты, если не оценивать их вовремя. Если вы определите реальные показатели по проекту, то в будущем окажетесь более подготовленными к постановке реалистичных целей и составлению выполнимых графиков работ.

Вторым шагом, необходимым для создания приложения, отвечающего определенному уровню качества, является включение постоянного инспектирования в процесс. По личному опыту автора, и для разработчиков, и для студентов становится сюрпризом то, насколько повышается эффективность их работы благодаря инспектированию. Однако существует и обратная сторона медали. Инспектирование требует значительных временных затрат на всем протяжении проекта, и эти затраты необходимо учитывать при составлении графика работ. Составление таких графиков обсуждается в главе 2.

Третий шаг касается спецификации желаемого уровня качества и процедур. Использование такого стандарта, как IEEE SQAP, предоставляет полный список проблем, которые необходимо решить. Написание и реализация SVVP обычно чересчур трудоемки для включения в односеместровый проект.

1.7. Управление документацией

1.7.1. Введение

Управление документацией программного проекта требует значительных организационных навыков, ибо документация — это сложный, живой организм, подверженный постоянным изменениям, которые зачастую вносятся одновременно множеством людей. Написание хорошей и гибкой документации во многом сродни написанию хорошего и гибкого программного кода.

Управление документацией подразумевает поддержание ее *полноты* и *согласованности* и включает в себя также *управление конфигурациями*. Вопроса *полноты* мы уже касались в разделе 1.5.2, где рассматривался типичный комплект документации, охватывающий процесс разработки и сопровождения. *Согласованность* означает, что набор документов не содержит внутренних противоречий. Проблема в том, что когда этот набор достаточно велик, то довольно сложно избежать появления в нем почти взаимоисключающих утверждений. Поддержка *конфигурации* — это координация различных версий и частей документации и программного кода. Согласованности и конфигурации документации посвящены следующие разделы.

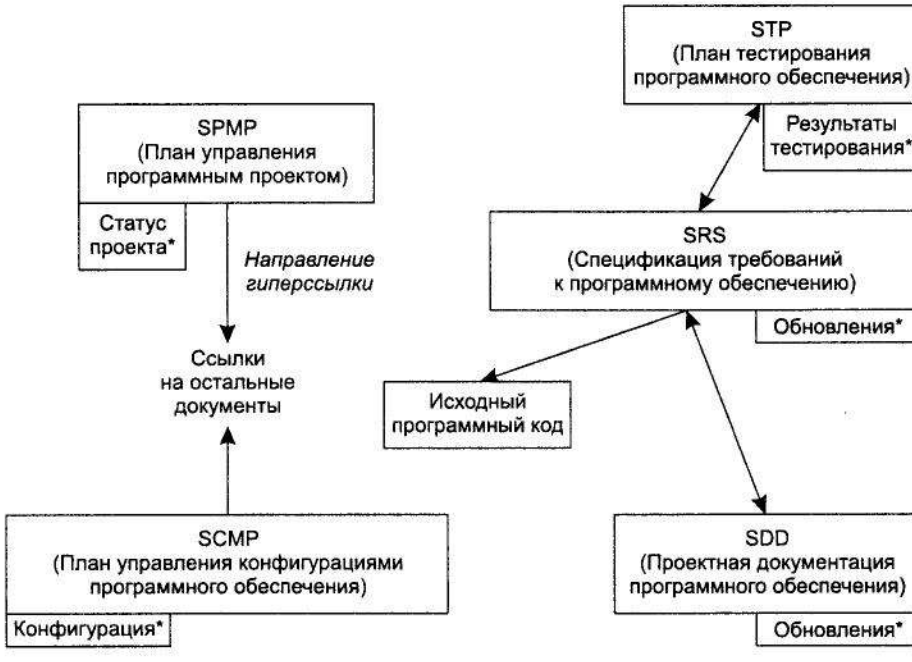
1.7.2. Согласованность и целостность документации

Ключ к поддержанию непротиворечивости состоит в том, что в документации каждая спецификация должна располагаться только в одном месте. Такую документацию мы называем *целостной* (single source documentation).

Для примера представим себе некоторое научное приложение с таким требованием: «счетчик числа бактерий является целочисленной переменной, принимающей значения от 1 до 100 000 000; эта переменная должна быть всегда доступна». Это требование документируется в SRS, и, на первый взгляд, выглядит заманчивым повторить его в разных других местах — скажем, в исходном коде (в заголовке подходящей функции) и в руководстве пользователя. Однако прежде чем так сделать, полезно оценить те проблемы, к которым приводят подобные повторения. А проблемы эти проистекают из неизбежных изменений, которым подвергается проект на протяжении всего своего жизненного цикла и которые требуется отражать в документации. Если потребовалось изменить наше требование, касающееся счетчика числа бактерий, это изменение нужно отразить везде, где оно повторяется. Отразить все изменения в спецификациях с отслеживанием всех мест, где эти спецификации повторяются, практически нереально, особенно если принять во внимание, что обычно это приходится делать в условиях сжатых

сроков выполнения проекта. Поэтому подобные повторения являются источником противоречивости в документации, а противоречия ведут к неудаче проекта. Разумеется, мы не можем избежать необходимости ссылаться из разных мест на один и тот же факт, но делать это нужно, не нарушая целостности документации, например, используя механизм гиперссылок.

Версия набора документации с гиперссылками, в которой части одного документа ссылаются на соответствующие части других документов, представлена на рис. 1.19. Например, поскольку каждое требование приводит к своему исходному коду, то исходный код может ссылаться на соответствующие требования в SRS. Технология гиперссылок позволяет проследить каждое требование до его реализации в коде и тестового примера. При этом требование ни разу не переопределяется.



Динамические компоненты

Рис. 1.19. Пример набора документации с гиперссылками

Динамические компоненты, отмеченные на рис. 1.19, — это те части документов, которые должны обновляться и (или) дополняться в ходе работы над проектом. Другими словами, большинство документов — «живые» объекты, о которых необходимо заботиться. Пример тому — SCMP, который описывает версии конкретного документа. При внесении дополнений и изменений в ходе итеративной разработки проекта часто проводится обновление SRS.

Иногда встречаются ситуации, в которых просто необходимо переопределить требование (представить его в другом виде). Например, руководства пользовате-

ля пишутся в несколько иной манере, нежели документация разработки. К тому же иногда требуется многоуровневая структура документации. Так, на одном уровне нужен только краткий обзор, тогда как на другом — все до малейших деталей. Усложнение структуры документации может привести к несогласованности ее ведения. Решением проблемы с несогласованностью в документации могут стать гиперссылки при условии поддержки перекрестных ссылок. Например, некоторый обзорный раздел может ссылаться на соответствующие разделы с подробными описаниями или требование из SRS может ссылаться на соответствующий раздел в руководстве пользователя. Это не избавляет от повторного изложения одного материала, но облегчает поддержку целостности, поскольку обе версии виртуально находятся «рядом». В главе 7 приведен пример целостной документации, где заголовки функций в исходном тексте программы ссылаются на соответствующие требования в SRS.

Для небольших организаций оказывается довольно трудно синхронизировать документооборот. Например, после принятия первоначального варианта требований разработчики часто добавляют некоторые допущения в оформлении и функциональности без обновления соответствующих требований. Упомянутая выше техника применения гиперссылок значительно облегчает решение проблем разработчиков. Однако ее применение не является гарантией успеха. Только путем обучения разработчиков программных продуктов и при понимании ими того, что ведение согласованной документации является их профессиональным долгом, можно добиться отличных результатов.

1.7.3. Управление конфигурациями

За время жизни проекты претерпевают изменения в двух направлениях. Во-первых, это приобретение новых частей, а во-вторых, получение новых версий существующих частей. Управление конфигурациями и есть управление упомянутыми частями проекта.

Незаконченное строительство дороги — хороший пример идущего инженерного проекта. Его легко увидеть, и уж конечно невозможно потерять! Мы же должны быть гораздо аккуратнее во время работы над программным продуктом, поскольку он или какая-то из частей как раз могут быть потеряны. Говоря о потере продукта или его части, мы имеем в виду не столько потерю самих файлов, сколько несоответствие их версий. Важнейшим требованием к ведению проекта является знание точного местонахождения частей проекта и связей, установленных между ними. *Части* проекта включают в себя не только исходный текст программ, но и всю документацию, в том числе план проекта. По этой причине мы столь рано начинаем говорить в этой книге об управлении конфигурациями. Мы должны уметь отслеживать изменения в документах еще до того, как разработан SPMP (глава 2).

Обычно в результате выполнения проекта создается несколько версий продукта. Например, существует несколько способов рендеринга сцен в видеоиграх, и мы можем попробовать некоторые из них. В итоге как минимум один из вариантов не попадет в конечный продукт. Аналогичная ситуация складывается

и с функциями. Часть из них отбрасывается из-за изменений в исходных требованиях, часть — из-за появления более полных версий. К тому же в работе над проектом может использоваться много вспомогательного программного обеспечения, которое не включается в конечный продукт. Все эти объекты должны отслеживаться, чтобы команда, работающая над проектом, имела четкое представление о том, что происходит в каждый конкретный момент времени. Наконец, когда проект выполнен, какие-то его части могут оказаться полезными для других проектов, а для этого нужно знать, где и как они могут быть получены.

1.7.3.1. Элементы конфигурации

Для отслеживания частей проекта сначала мы должны определить их границы. Элементы конфигурации (CIs — configuration items) — это части проекта, отслеживаемые системой управления конфигурациями. Например, как показано на рис. 1.20, шестая версия метода расчета социальных отчислений для модуля подготовки платежных ведомостей некой бухгалтерской системы может быть названа элементом конфигурации S6.

Элемент конфигурации может состоять из других элементов конфигурации. Так, на рис. 1.20 элемент «Платежная ведомость версии 0.3.4.2» — модуль подготовки платежных ведомостей бухгалтерской системы — состоит из версии 6 части S, версии 1 части A, версии 3 части E и т. д. В первом примере метод расчета выплат S обновляется с S6 до S7. Во втором к системе добавляется новый метод F1. Система конфигураций должна уметь различать все эти версии. На рис. 1.20 видно, что для этого изменяется номер версии модуля. Обычно классы — это отдельные элементы конфигураций. Отдельные функции могут быть элементами конфигураций, хотя это и не типичный случай. Мы не будем использовать элементы конфигураций мельче, чем функции, так как отслеживание составляющих частей функции практически неосуществимо. Значимые наборы данных, такие как глобальные таблицы, тоже могут быть элементами конфигураций.

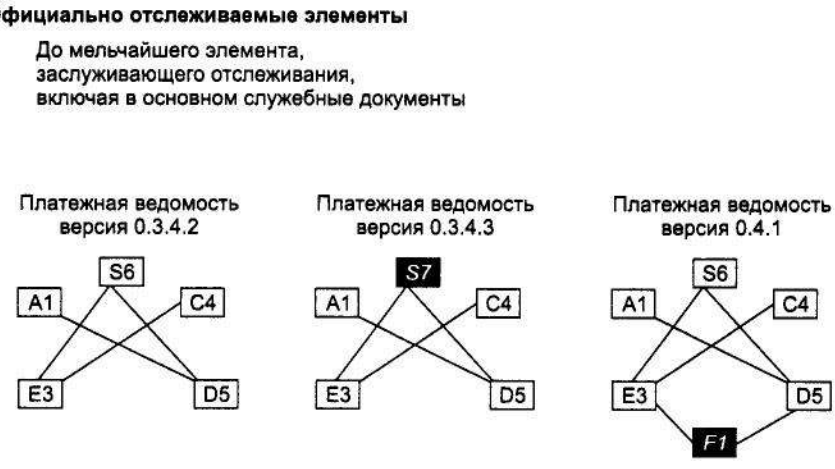


Рис. 1.20. Элементы конфигурации

Далее необходимо организовать использование элементов конфигураций всеми сотрудниками, участвующими в проекте. Если не проделать это достаточно аккуратно, то может получиться настоящий хаос. Приведем несколько примеров неверных действий.

- Разработчик программного обеспечения берет копию класса *Заказчик*, чтобы изменить ее и сделать более функциональной. Он заканчивает свою работу, регистрирует ее и заменяет оригинальную версию. Позже выясняется, что с новой версией есть проблемы, и принимается решение использовать старую версию, пока новый класс *Заказчик* дорабатывается. Однако другие части проекта уже изменены для работы с новой версией этого класса.

  Старая версия должна быть всегда доступна и должно быть четко указано, какие части приложения работают со старой версией, а какие уже изменены для нового варианта. Такая организация позволяет при необходимости сделать откат к предыдущей версии. В то же время нужно восстановить и описание требований, которым отвечает старая версия.

- Разработчики программного обеспечения Абель и Берил взяли копии класса *Заказчик*, чтобы модифицировать его. Абель завершает работу и замещает старый класс *Заказчик* новым. Берил завершает работу и замещает старый класс *Заказчик* своим новым, теряя при этом все изменения, внесенные Абелем.

  Системы управления конфигурациями позволяют распределять доступ к элементам конфигураций. Таким образом, элемент конфигурации может быть получен для изменения только одним разработчиком, в то время как остальные инженеры могут работать с копией только для чтения.

Обычно системы управления конфигурациями удовлетворяют следующим минимальным требованиям.

- Деятельность по определению элементов конфигурации.
- Отказ в праве на модификацию — для предотвращения одновременной работы более чем одного человека над элементом конфигурации.
- Авторизация доступа с правом модификации — дополнительно (не обязательно).
- Процедура включения в проект:
  - процесс авторизации;
  - привлечение тестирования и т. д.
- Запись о предыдущих группировках согласующихся элементов конфигурации.

Например, должны существовать четкие процедуры, в ходе которых получается доступ к элементам конфигурации для их изменения, добавляются новые элементы конфигурации и модифицированные элементы конфигурации возвращаются в проект. Должна вестись документация и проводиться тестирование.

В частности, необходим ясный процесс авторизации для включения модифицированных документов обратно в проект. Контролировать авторизацию может один человек, например руководитель проекта, или группа, обычно называемая группой управления изменениями.

1.7.3.2. План управления конфигурациями (SCMP): стандарт IEEE 828-1990

IEEE разработал стандарт по планированию управления конфигурациями, IEEE 828-1990. Этот стандарт может быть очень полезен в процессе управления конфигурациями при проверке того, охвачены ли все основные положения. Содержание стандарта представлено далее.

1. **Введение**
2. **Управление конфигурациями**
   - 2.1. Организация
   - 2.2. Ответственность за управление конфигурациями
   - 2.3. Применяемые политики, директивы и процедуры
3. **Виды деятельности**
   - 3.1. Определение конфигурации
     - 3.1.1. Определение элементов конфигурации
     - 3.1.2. Именование элементов конфигурации
     - 3.1.3. Получение элементов конфигурации
   - 3.2. Контроль конфигурации
     - 3.2.1. Запрос на изменения
     - 3.2.2. Оценка изменений
     - 3.2.3. Одобрение или неодобрение изменений
     - 3.2.4. Реализация изменений
   - 3.3. Определение статуса конфигурации
   - 3.4. Аудиты и обзоры конфигурации
   - 3.5. Контроль интерфейса
   - 3.6. Контроль поставщиков и субподрядчиков
4. **Расписание**
5. **Ресурсы**
6. **Сопровождение**

В разделе 3.3 SCMP задокументировано, каким образом определяется статус управления конфигурациями (например: письменно, раз в неделю). Раздел 3.6 присутствует, если используется специальный инструментарий для управления конфигурациями или если управление конфигурациями выполняется субподрядчиком. Стандарт IEEE подробно описывает назначение каждого

из приведенных разделов. Мы пользуемся IEEE 828-1990 в примере в конце этой главы.

Широко используются коммерческие системы управления конфигурациями, такие как SourceSafe компании Microsoft.

ОДИН ИЗ СПОСОБОВ ПЛАНИРОВАНИЯ УПРАВЛЕНИЯ КОНФИГУРАЦИЯМИ

1. Составьте наброcок вашего SCMP:
   - установите порядок действий при внесении изменений;
   - не указывайте ссылки на инструменты, если они еще не определены;
   - обратитесь к примеру в конце этой главы.
2. Определитесь с тем, какие из инструментов управления конфигурациями вам необходимы.
   - В учебных заданиях достаточно блокировки и резервного копирования.
3. Оцените инструменты с точки зрения ваших потребностей и бюджета:
   - широко используются коммерческие инструменты;
   - попробуйте применить бесплатные веб-сайты для хранения документации, попробуйте простой способ организации доступа для модификации, например переименование.
4. Завершите ваш SCMP.

Для учебных команд вполне подойдут веб-сайты, такие как www.egroups.com, обеспечивающие хранение документации. Одним из простых способов систематизации доступа для модификации является изменение типа документа. Например, когда Джо берет SQAP на модификацию, то имя файла меняется с SQAP.txt на SQAP.joe. Хотя управление конфигурациями относится и к документации, и к исходному коду, соглашение об именовании файлов обычно планируется отдельно. Например, мы не можем заменить myClass.java на myClass.joe, не нарушив при этом порядок компиляции. Некоторые группы поддерживают два комплекта файлов. В одном комплекте содержится текущая версия, являющаяся основой, которая может быть изменена только в ходе формального процесса. Другой комплект содержит версии, находящиеся в разработке.

Различные инструменты управления конфигурациями, такие как FtpVC, доступны для бесплатного использования. Убедитесь, что ваш процесс не основан на постоянном ручном вмешательстве и что в нем нет узких мест с чрезмерной перегрузкой одного человека. При оценке какого-либо инструмента убедитесь, что время на обучение окупится полезностью этого инструмента. Помните, что существует множество вариантов различных способов управления конфигурациями помимо использования своих инструментов. Какой бы вариант вы ни выбрали, проверьте его сначала на тестовом примере. Убедитесь, что процесс проходит гладко, и только тогда вы можете быть спокойны за процесс управления конфигурациями на этапе реализации, когда время ограничено.

Студенческие команды создают многочисленные вполне работоспособные системы, основанные на доступных бесплатных инструментах. Однако в рабочих проектах требуются профессиональные инструменты управления конфигурациями.

1.8. Введение в методы оценки возможностей

Разработка программного обеспечения стала восприниматься как одно из ценнейших качеств и возможностей организаций. Однако тут же возникает вопрос: «Как оценить эту возможность?», который в свою очередь приводит к вопросу: «Насколько мы готовы?». Немногие организации могут ответить на этот вопрос «Отлично».

Этот раздел описывает варианты оценки возможностей инженеров, команд и организаций в программных разработках. Уоттс Хэмфри и Институт технологий разработки программного обеспечения (SEI) разработали *Индивидуальный процесс разработки программного обеспечения*, *Командный процесс разработки программного обеспечения* и *Модель зрелости возможностей*, чтобы оценить эти уровни возможностей. Данный раздел описывает каждый из этих вариантов и их взаимосвязи.

Разработка программного обеспечения может быть рассмотрена с точки зрения самостоятельного разработчика (раздел 1.8.1), команды (раздел 1.8.2) или целой организации (раздел 1.8.3).

1.8.1. Введение в Индивидуальный процесс разработки программного обеспечения (PSP)

Хэмфри весьма искусно определил, какими конкретно навыками должен обладать компетентный иженер — разработчик программного обеспечения. Они описаны в *Индивидуальном процессе разработки программного обеспечения* (PSP — Personal Software Process), который «предоставляет детальные описания методов планирования и оценки, показывает разработчикам, как измерять собственную продуктивность и соотносить ее с существующим планом, и объясняет, почему описанные методы могут помочь им в их работе» [52]. Предполагается, что PSP помогает выработать определенные рабочие навыки, особенно связанные с количественной оценкой работы. (Как много времени я провел над этим кодом? И сколько написал строк? А сколько ошибок в них допустил? И так далее.) При этом PSP предполагает, что инженер уже обладает знаниями непосредственно в области программирования.

PSP делится на стадии постепенного роста, названные PSP0, PSP1, PSP2 и PSP3 [49] (рис. 1.21).

- **PSP0: Базовый процесс.**

  PSP0 разрешает студенту пользоваться собственными методами разработки, но требует от него:

  - записывать время, затраченное на работу над проектом;
  - записывать найденные дефекты;
  - записывать типы дефектов.

  PSP0 дополняется PSP0.1, который предписывает студенту выбрать:

  - стандартное определение понятия «строка кода»;
  - способ для указания того, как он мог бы улучшить свои навыки разработчика.

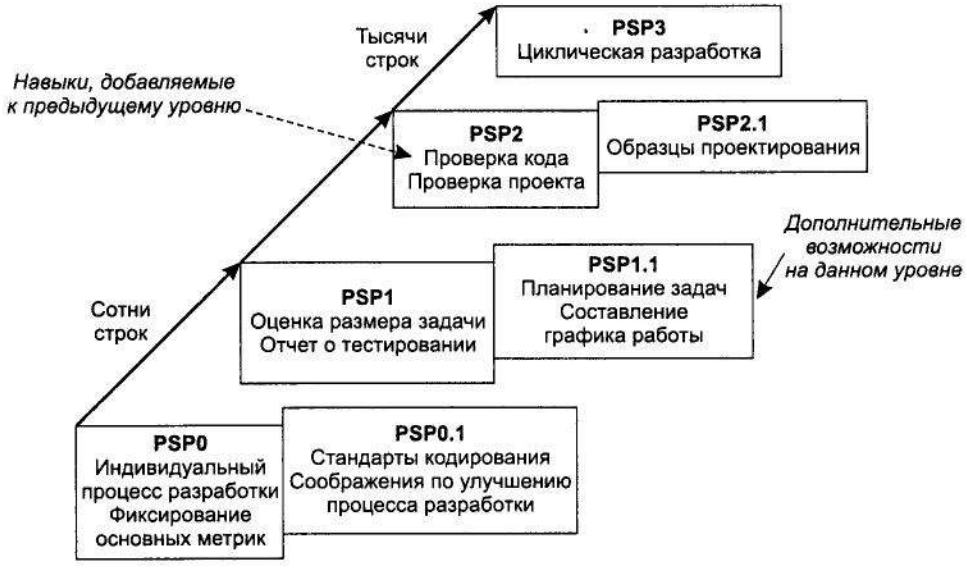


Рис. 1.21. Эволюция PSP

- **PSP1: Индивидуальный процесс планирования.**

  PSP1 призван помочь инженеру полнее понять отношение между размером программы и временем, которое он тратит на ее разработку. PSP1 должен сформировать *упорядоченную систему*, в рамках которой инженер может производить оценки, выполнять задачи, проверять статус работы и записывать результаты. Всем разработчикам ПО постоянно задается один и тот же вопрос: «Сколько времени у вас это займет?». PSP помогает инженерам научиться реалистично отвечать на этот вопрос.

  PSP1 добавляет следующие требования к PSP0:

  - способность оценивать размер задачи;
  - систематический подход к описанию результатов тестирования.

  PSP1 дополняется PSP1.1, который описывает способности:

  - планировать программные задачи;
  - распределять их по времени и составлять график работы.

- **PSP2: Индивидуальный процесс контроля качества.**

  PSP2 разработан, чтобы помочь инженерам объективно и реалистично справляться с программными дефектами. Идея заключается в том, чтобы обнаружить и исправить максимальное количество дефектов перед сдачей работы до формальной проверки-инспектирования (см. раздел 1.6.4).

  PSP2 включает в себя:

  - индивидуальную проверку проекта и архитектуры;
  - индивидуальную проверку кода.

PSP2 дополняется PSP2.1, который включает:

- набор контрольных вопросов для проверки целостности избранных программных решений.

✦ **PSP3: Циклический индивидуальный процесс.**

PSP3 предназначен для применения процесса PSP к большим программным модулям (содержащим тысячи строк кода) путем разбиения больших программ на меньшие этапы.

PSP3 описывает:

- способ применения PSP к каждому этапу, что позволяет получить высококачественный результат, пригодный для следующей итерации;
- *регрессионное тестирование*, которое призвано проверять, что тесты, разработанные для предыдущих итераций, успешно выполняются и на новых этапах.

1.8.2. Командный процесс разработки программного обеспечения (TSP)

В 1999 году Уоттс Хэмфри представил положительные результаты в постановке целей и процедур зрелости для командной разработки программ. Он назвал этот процесс *Командный процесс разработки программного обеспечения* (TSP – Team Software Process [50]). Перечислим задачи TSP.

✦ Собрать самоуправляемые команды:
- 3–20 разработчиков;
- установить собственные цели;
- составить свой процесс и планы;
- отслеживать работу.

✦ Показать менеджерам, как управлять командами:
- инструктаж;
- мотивация;
- поддержка максимальной производительности.

✦ Ускорить продвижение по шкале CMM:
- сделать пятый уровень CMM нормой.

✦ Обеспечить пути улучшения для высокоразвитых организаций.

✦ Содействовать университетскому образованию для команд промышленного уровня.

Ставка TSP на командную инициативу поддерживает высокую степень профессионализма среди программных разработчиков. Например, Хэмфри утверждает, что соглашаться с планами и расписаниями, которые не могут быть выполнены, является для инженеров признаком непрофессионализма даже в тех случаях, когда требуется так поступать. В такой ситуации он предлагает вести перегово-

ры. Профессионализм, напоминает Хэмфри, возлагает обязательство служить на благо общества, а не только на благо работодателя. Также заслуживает внимания упор TSP на внешнее управление командой. Управление предназначено не только для раздачи указаний и утверждения последних сроков, но и для обеспечения руководства, соответствующих инструментов и других необходимых ресурсов. Мы вернемся к TSP при обсуждении управления проектом в следующей главе.

1.8.3. Модель зрелости возможностей (CMM)

Возможность организации разрабатывать качественное программное обеспечение зависит от множества факторов, как организационных, так и человеческих. Человеческие факторы варьируются от индивидуальной компетентности (рассматривается в PSP) до управления процессом. Улучшение процесса в организации лучше всего представлять по этапам.

В 80-х годах Институт технологий разработки программного обеспечения (SEI) от имени Министерства обороны США установил простую классификацию возможностей субподрядчиков. План Министерства обороны состоял в том, чтобы давать оборонные заказы только таким подрядчикам, которые обладают определенным уровнем возможностей. Система, разработанная SEI, известна как *Модель зрелости возможностей* (CMM – Capability Maturity Model). CMM оказалась очень эффективной для определения уровня компетентности в разработке программного обеспечения. Многие организации, как военные, так и гражданские, использовали CMM для измерения своих усилий по улучшению процесса разработки. Например, вместо того чтобы характеризовать организацию как «достаточно хорошую» в области разработки программного обеспечения, можно точно указать, что организация «находится на уровне 3 по CMM». Среди известных автору организаций, которые намерены улучшить свои показатели по CMM, есть и мощные компании военно-промышленного комплекса, и компании, занимающиеся программным обеспечением проверки орфографии.

CMM классифицирует организации по пяти уровням следующим образом.

- **Уровень 1: Начальный.** Уровень 1 соответствует самому примитивному статусу организации. На этом уровне признается, что организация способна разрабатывать программное обеспечение. Организация не имеет явно осознанного процесса, и качество продукта целиком определяется индивидуальными способностями разработчиков. В типичном случае один из членов команды проявляет инициативу – что делать дальше, и команда следует его указаниям. Успех одного проекта не гарантирует успеха следующего (за исключением случая, когда состав команды не меняется и проект аналогичный). По завершении проекта не фиксируются данные о трудозатратах, расписании и качестве.

  Процесс: не определен; зависит от конкретной задачи.

  Результат: зависит от индивидуальных способностей.

  Недостаток: нет четко заданного процесса.

- **Уровень 2: Повторяемый.** Уровень 2 соответствует организациям, которые до некоторой степени отслеживают свой процесс работы. Поддерживаются записи о трудозатратах и планах. Функциональность каждого проекта описана в письменной форме. Следовательно, возможно оценить затраты на разработку аналогичного проекта той же командой. Это заметное улучшение по сравнению с уровнем 1. Для дальнейшего улучшения необходимо иметь возможность оценивать затраты независимо от наличия или отсутствия конкретных людей в команде проекта. В середине 1999 года только 20 % организаций имели уровень 2 или выше.

  Процесс: ведется документация и учет трудозатрат, отслеживается ход выполнения планов и функциональности (постфактум).

  Результат: воспроизводим только для аналогичных проектов.

  Недостаток: нет полного процесса.

- **Уровень 3: Установленный.** Уровень 3 соответствует организациям, которые имеют определенный, документированный и установленный процесс работы, не зависящий от отдельных личностей. Обычно это один из процессов, описанных в данной главе: водопадный, спиральный или инкрементальный. Некоторые организации используют существующие стандарты IEEE, в то время как другие определяют свои собственные корпоративные стандарты. Грубо говоря, как только руководство вводит в действие согласованные профессиональные стандарты, а разработчики их выполняют, организация достигает уровня 3. Обычно это предполагает проведение специального обучения персонала. Допускается настройка стандартного процесса для конкретной команды применительно к конкретным обстоятельствам. Хотя организации уровня 3 в состоянии производить программное обеспечение гарантированного качества и таких организаций относительно немного, им все же не хватает возможности точно предсказывать затраты на проект. Такие организации в состоянии достаточно надежно сделать предсказание только для проектов, которые полностью аналогичны уже выполненным ранее.

  Процесс: документированный; стандартный; настраиваемый.

  Результат: целостность.

  Недостаток: результаты не вполне предсказуемы.

- **Уровень 4: Управляемый.** Уровень 4 соответствует организациям, которые могут точно предсказать сроки и стоимость работы. Один из способов, которым это делается, состоит в том, чтобы классифицировать работы и их части и измерять и сохранять данные о стоимости и трудозатратах на проектирование и реализацию этих частей. Такие измерения накапливаются в базе данных, и на их основе делаются оценки новых работ. Как отмечает Хэмфри, это еще не «ракетные технологии», но это требует значительных организационных возможностей.

  Кажется, что уровень 4 — это предел возможностей, но это не так. Мы знаем, что технологии программирования меняются быстро. Например, объектно-ориентированная парадигма привнесла значительные изменения в техно-

логию: повторное использование и концепция компонентов оказывают все возрастающее влияние на разработку. Но появление новых парадигм и идей непредсказуемо. Таким образом, возможности организации уровня 4, которые не меняются в соответствии с изменениями обстановки, постепенно уменьшаются.

Процесс: детальные измерения и управляемость.

Результат: продукт и процесс с предсказуемой количественной оценкой качества.

Недостаток: нет механизма улучшения процесса.

- **Уровень 5: Оптимизированный.** Вместо того чтобы пытаться предсказывать новые изменения в технологиях, гораздо лучше установить постоянно действующую процедуру поиска и освоения новых и улучшенных методов и инструментов. Таким образом, уровень 5 соответствует организациям, в которых имеется встроенный процесс улучшения процесса. Другими словами, их процесс (можно сказать, метапроцесс) включает в себя процесс постоянного улучшения текущего процесса разработки в организации на основе постоянной оценки текущего процесса, поиска новых средств и технологий и включения их в текущий процесс. Этой впечатляющей возможностью в начале 2000 года обладали только несколько организаций. Однако со времени появления CMM уровень 5 постепенно превращается из недостижимой мечты в конкретную цель.

  Процесс: непрерывное улучшение процесса за счет измеряемых количественных показателей; расширяемые возможности; инновационные идеи и технологии.

1.8.4. Связь между PSP, TSP и CMM

В [51] Хэмфри связывает три модели возможностей, которые обсуждались в предыдущих разделах. Сравнение этих моделей приводится в табл. 1.4. В качестве примера взаимосвязи PSP, TSP и CMM отметим, что программы обучения относятся только к CMM: поскольку в них вовлекается вся организация, они не могут быть частью TSP или PSP. Другой пример заключается в том, что для программной конфигурации требуется работа в команде, и она никак не предназначена для персональной программной разработки.

Таблица 1.4. Связь между PSP, TSP и CMM

Уровень CMM	В центре внимания	Ключевая область процесса	PSP	TSP
5. Оптимизированный	Непрерывное улучшение процесса	Предупреждение появления дефектов	×	×
		Управление сменой технологии	×	×
		Управление сменой процесса	×	×

продолжение

Таблица 1.4 (*продолжение*)

Уровень CMM	В центре внимания	Ключевая область процесса	PSP	TSP
4. Управляемый	Качество продукта и процесса	Управление процессом на основе количественных показателей	×	×
		Управление качеством программного обеспечения	×	×
3. Установленный	Процесс разработки	Ядро организационного процесса	×	×
		Определение организационного процесса	×	×
		Программы обучения		
		Интегрированное программное управление	×	×
		Разработка программного продукта	×	×
		Координация внутри группы		×
		Коллегиальные совещания	×	×
2. Повторяемый	Управление проектом	Управление требованиями		×
		Планирование программного проекта	×	×
		Отслеживание программного проекта	×	×
		Контроль качества программного обеспечения		×
		Управление конфигурациями программного обеспечения		×
		Контроль подрядчиков		

1.9. Подведение итогов

Создание программных продуктов является трудным делом. Водопадный процесс является простейшим типом процесса; в то же время он служит базовым процессом для спиральной (или итеративной) разработки. Третий из основных типов процесса — инкрементальная разработка. Успех зависит от профессионального взгляда с ориентацией на качество, всесторонней поддержки самостоятельных разработчиков и команд (например, через PSP/TSP) и выполнения полноценного, стандартизированного процесса разработки (как это приведено в CMM). Идеальный процесс разработки включает в себя и постоянное оценивание и улучшение. Подведем итоги и укажем типичный набор документации проекта.

- Разработка программного обеспечения — это очень серьезное занятие.
- Основные модели процессов:
  - водопадная, спиральная, инкрементальная.
- База возможностей: CMM, TSP, PSP.
- Качество — это отличительная черта профессионализма:
  - метрики, которые необходимо задать;
  - повсеместное инспектирование;

- тщательное тестирование;
- непрерывное улучшение процесса разработки.

✦ Документация: SCMP, SVVP, SQAP, SPMP, SRS, SDD, STP, программный код, руководство пользователя.

Упражнения

Ответы и подсказки для упражнений, помеченных символами «о» или «п», приводятся в конце этой главы.

Вопросы для проверки

П1.1ⁿ.

1. Перечислите четыре составляющие производства программного обеспечения.
2. Являются ли некоторые из них более важными, нежели другие?

П1.2°. (Для ответов используйте текст книги.)

1. Назовите четыре основных фазы в разработке программного продукта.
2. Что такое водопадный процесс?
3. Каким образом водопадный процесс применяется в современном процессе разработки?

П1.3°. Назовите кроме водопадного еще два процесса разработки программного обеспечения.

П1.4°.

1. В чем заключается разница между верификацией и валидацией? (Для ответа см. раздел 1.6.6.)
2. Каковы достоинства и недостатки составления плана экспертизы (верификации и валидации) до составления плана ведения открываемого проекта?

П1.5°. Приведите два преимущества и два недостатка использования стандартов для документации.

П1.6°.

1. Что такое метрики?
2. Почему метрики полезно использовать?

П1.7°.

1. Назовите методы оценки процесса индивидуальной, командной и организационной разработки, созданные SEI и Хэмфри.
2. Кратко опишите эти методы. Обозначьте их основные составляющие и цели. (Ответ: раздел 1.8.)

3. Убедитесь в том, что вы рассказали обо всех уровнях CMM в предыдущем вопросе.

Упражнения в команде

Перед каждым упражнением решите всей группой, как вы будете его проводить, проверьте подсказки, приведенные ниже, а затем выполните задания.

К1.1[п]. (Тема: «План управления конфигурациями».)

Выработайте план управления конфигурациями программного обеспечения для вашего текущего проекта, используя стандарт IEEE 828-1990. Пример в конце этой главы будет помогать вам в этом; однако ваш документ будет более индивидуальным, отражая доступные вам ресурсы. Не включайте материал, который не способствует достижению целей создания этого документа. Избегайте узких мест и лишних процедур. Проверьте ваш план, используя его для создания необходимых документов в упражнении К1.2, приведенном ниже.

Перед тем как начать, оцените то количество дефектов на страницу, которое команда ожидает найти в ходе заключительного обзора. Поддерживайте отслеживание и ведите учет времени, затрачиваемого каждым из членов команды и целой командой. Определите реальную плотность дефектов (среднее число дефектов на страницу). Оцените эффективность деятельности вашей команды на каждом этапе по десятибалльной шкале. Подведите итоги, используя численные результаты. Подумайте, как можно было бы улучшить командный процесс.

Критерии оценки.

1. Практичность. Насколько можно быть уверенным в том, что документы и их версии будут сохранны, доступны и согласованы? («Отлично» — наиболее вероятно.)
2. Конкретность. Насколько конкретен план в смысле указания ролей и мест хранения? («Отлично» — каждый разработчик точно знает, что делать.)
3. Оценка и улучшение процесса. До какой степени команда понимает сильные и слабые стороны своего процесса? Насколько конкретны планы по улучшению процесса? («Отлично» — полное понимание, планы конкретны и реалистичны.)

К1.2[п]. (SVVP.)

Выработайте необходимые части SVVP, используя стандарт IEEE 1012-1986. Измерьте и приведите метрики, упомянутые в задаче К1.1.

Критерии оценки.

1. Практичность. Насколько план обеспечивает достаточную валидацию и верификацию проделанной работы? («Отлично» — вполне практично в среде проекта.)
2. Конкретность. Насколько конкретен план в смысле указания процедур и участников? («Отлично» — точно описывает, как выполняется V&V в среде проекта.)

3. Отношение участников. В какой степени участники проекта воспринимают ваш план как помощь? («Отлично» — написан с учетом интересов разработчиков.)

Подсказки

П1.1.

1. Четыре составляющие начинаются на букву «П». (Ответ приведен ниже.)

К1.1 и К1.2. Не углубляйтесь в разделы, которые вам еще не знакомы. Добавляйте только те части, которые можно сделать за семестр. Добейтесь реалистичности плана.

Ответы

П1.1. 1 и 2. Персонал, Процесс, Проект и Продукт. Все они важны. По мнению автора, ни один из них не является более важным, чем другие.

П1.2.

1. Анализ требований, проектирование, реализация, интеграция и тестирование.

П1.3. Спиральный и инкрементальный процессы.

П1.4.

2. Среди преимуществ — то, что процедуры контроля качества могут быть специфицированы сразу для всей организации, гарантируя тем самым унифицированность продуктов организации. Это позволило бы сэкономить время на переделке и адаптации процедур из проекта в проект. Недостатком является то, что невозможно предусмотреть все детали данного плана до момента утверждения плана проекта. (Например, мы не можем составить расписание верификации и валидации.)

П1.5. Стандарты документации экономят наше время на обдумывание структуры документа. Они напоминают о важных вещах, которые можно упустить. Использование стандартов унифицирует документацию в организации. Недостаток использования стандартов заключается в том, что разработчики иногда чувствуют себя обязанными написать «хоть что-нибудь» в стандартный раздел документа, в то время как возможно, что этот раздел стандарта неадекватен данному конкретному случаю. В результате возникает ощущение бесплодной «бумажной» работы. С другой стороны, разработчики несколько скованы стандартом и стесняются добавить нестандартный раздел в документ, хотя, может быть, он был бы очень важен в конкретном случае (например, стандарты IEEE не содержат некоторых разделов, очень полезных при разработке веб-приложений).

П1.6. Метрики — это количественные значения показателей процесса разработки. Они важны, поскольку вносят точность в процесс разработки, превращая его из шаманства в инженерную деятельность.

П1.7. CMM для организаций, TSP для команд и PSP для разработчиков.

Пример 1. План управления конфигурациями (SCMP)

[Примечание для студентов. Неплохо, если каждый участник команды будет подписывать документы, которые готовит. Это концентрирует внимание и повышает чувство ответственности за содержание документов.]

Утверждаю
_____Дата_____
Содержание данного SCMP следует стандарту IEEE 828-1990
01.05.98 Э. Брауде: Создание первой версии
10.01.99 Р. Боствик: Рецензирование
18.01.99 Э. Брауде: Расширен раздел 5.2
18.05.99 Э. Брауде: Проверка для выпуска
30.04.99 Э. Брауде: Окончательное форматирование
02.05.99 Э. Брауде: Выпуск

1. Введение

Данный План управления конфигурациями ПО (SCMP) описывает, как ведется работа с артефактами игры *Встреча*.

1.1. Сокращения

CI — Configuration Item. Элемент конфигурации — любой элемент, отслеживаемый системой управления конфигурациями.

CM — Configuration Management. Управление конфигурациями — процесс поддержки релевантных версий артефактов проекта.

SCMP — Software Configuration Management Plan. Данный документ.

1.2. Термины

Утвержденный CI — CI, подписанный руководством проекта.

Артефакт — окончательный или промежуточный материал проекта (например, документ, исходный код, объектный код, результат теста).

Главный файл — специальным образом построенный файл для данного проекта, определяется в разделе 3.1.2.

2. Управление конфигурациями

2.1. Организация

[Примечание для студентов. Определите, как должно быть организовано управление конфигурациями. Укажите роли, но не имена конкретных людей — для этого служат другие разделы.]

Специальный инженер, выделяемый отделом контроля качества, будет назначен *ведущим конфигурацию* на все время проведения данного проекта.

2.2. Ответственность за управление конфигурациями

[Примечание для студентов. Определите ответственность каждой роли. Если этого не сделать, то некоторые действия не будут выполняться, а другие будут выполняться несколькими членами команды больше чем один раз. Определите, кто является заместителем по каждой роли.]

2.2.1. Ведущий конфигурацию

[Примечание для студентов. Ведущий конфигурацию не обязательно должен делать всю работу сам. Скорее, он должен организовать работу и проследить за тем, чтобы она была выполнена.]

Ведущий конфигурацию отвечает за организацию и управление конфигурациями. Если это возможно, ведущий конфигурацию должен обсуждать планы управления конфигурациями с командой разработчиков до того, как эти планы вводятся в действие. Ведущий конфигурацию поддерживает данный документ (SCMP). Ведущий конфигурацию отвечает за установку и сопровождение инструментов управления конфигурациями, определенных в разделе 2.3. Архивация данных должна производиться в соответствии с корпоративной инструкцией 12345.

Ведущий конфигурацию отвечает за настройку, сопровождение и резервное копирование используемых инструментов управления конфигурациями. Он должен также разработать план действий на случай, если используемые инструменты окажутся неподдерживаемыми (например, по вине поставщика).

Дополнительные обязанности ведущего конфигурацию описаны в разделах 3.3, 3.4, 3.5 и 3.6.

2.2.2. Лидер проекта

Лидер проекта и руководитель проекта могут выполнять функции ведущего конфигурацию только в исключительных обстоятельствах. Они обязаны знать все соответствующие средства доступа к документам во время проведения проекта. Лидер проекта обязан проверить, что архивирование данных ведется в соответствии с инструкций, упомянутой в разделе 2.3.

Дополнительные обязанности лидера проекта описаны в разделах 3.3 и 3.4.

2.2.3. Разработчики

Каждый разработчик обязан выполнять правила управления конфигурациями, опубликованные ведущим конфигурацию. Разработчики также обязаны следовать документу 56789 «Должностные обязанности инженеров».

Дополнительные обязанности разработчика описаны в разделе 3.

2.3. Применяемые политики, директивы и процедуры

[Примечание для студентов. Такая деятельность, как управление конфигурациями, обычно осуществляется в соответствии с общей политикой организации. Студенческая команда должна идентифицировать и перечислить в этом разделе соответствующие правила и инструкции. При этом пункт 3 следует включить обязательно.]

1. Управление конфигурациями для данного проекта должно осуществляться в соответствии с указаниями по управлению конфигурациями, изложенными в корпоративном документе 7890 версии 6 от 15.08.98.

2. В соответствии с политикой улучшения процесса разработки требуется проводить обзорные совещания по ходу и в конце проекта, и все предложения по улучшению должны фиксироваться с целью использования их в организации. Данные совещания требуются для подготовки подразделения разработки к сертификации по CMM уровня 5. Результаты самооценки должны быть отправлены управляющему, ответственному за улучшение процесса, не позднее трех недель после того, как состоялось совещание. Все разделы, в которых указаны возможные улучшения, должны содержать соответствующий материал и конкретные примеры.
3. Все текущие и предшествующие версии CI должны сохраняться.
4. К главному файлу (определен в разделе 3.1.2) имеет доступ только ведущий конфигурацию, а в его отсутствие — начальник отдела.
5. Пароли управления конфигурациями должны меняться в соответствии с принятыми корпоративными правилами безопасности со следующим добавлением: никакой пароль не должен изменяться, пока лидер проекта, руководитель проекта и ответственный за качество не оповещены об изменении и не подтвердили получение оповещения.
6. Лидер проекта и начальник отдела должны всегда иметь полный доступ ко всем документам, которые затрагивает управление конфигурациями. Каждые две недели лидер проекта должен предоставлять форму http://www.ultracorp.division3.accessVerification своему начальнику для верификации прав доступа.
7. В проекте *Встреча* будет использоваться средство *SuperCMTool* версии 3.4, поставляемое компанией SuperCMTool.

   [Примечание для студентов. Все названия выдуманы.]
8. Архивация должна производиться в соответствии с правилами отдела, документ 123456.

3. Виды деятельности

3.1. Определение конфигурации

[Примечание для студентов. В этом разделе определяется, как создаются элементы конфигурации (CI) и как им назначаются имена. Если не определить и не выполнять такую процедуру, хаос неизбежен.]

3.1.1. Определение элементов конфигурации

Лидер проекта несет ответственность за определение всех элементов конфигурации. Разработчики, желающие определить новый CI, должны получить согласие лидера проекта по электронной почте или иным способом. Если лидер проекта недоступен в течение более чем одного рабочего дня, ведущий конфигурацию имеет право включить в конфигурацию предлагаемый элемент.

3.1.2. Именование элементов конфигурации

Ведущий конфигурацию несет ответственность за маркировку всех CI. Соглашение об именах следующее.

Корневой каталог: Encounter.

Вложенные каталоги: SRS или SDD или...

Файл с именем N_N_N.xxx соответствует версии N.N.N документа.

Например, версия SRS 2.4.8 имеет имя Encounter/SRS/2_4_8.txt.

Главный файл с именем Master находится в корневом каталоге и содержит информацию о текущем состоянии и предыдущих состояниях проекта. Например, файл Master может включать такую информацию.

Текущая версия проекта *Встреча* 3.7.1. Она включает версию 2.4.8 SRS и версию 1.4 SDD.

Предыдущая версия проекта *Встреча* 3.6.11. Она включает версию 2.4.8 SRS и версию 1.3 SDD.

Эта информация должна быть представлена в форме следующей таблицы.

Версия проекта Встреча	Версия SRS	Версия SDD	...
...	...	...	...

3.1.3. Получение элементов конфигурации

[Примечание для студентов. Составляя данный раздел, примите во внимание наиболее стрессовую часть проекта, то есть фазу реализации, когда несколько человек будут работать параллельно. Процесс доступа должен быть упорядочен, но он также должен позволять разработчикам быстро обращаться к различным частям проекта.]

Для модификации CI разработчик должен *взять* CI с помощью процедуры *checkout* инструмента SuperCMTool. Отметим, что SuperCMTool предлагает пользователю заполнить специальную форму и указать, на какой срок берется CI. Эта информация запоминается и предоставляется другим пользователям, желающим взять данный CI на модификацию. Любой желающий взять данный CI должен договориться с текущим владельцем CI о передаче средствами SuperCMTool. *Ни при каких обстоятельствах разработчики не должны передавать друг другу CI прямо, в обход SuperCMTool.* Для любого разработчика любой CI должен быть доступен на чтение в любое время.

3.2. Контроль конфигурации

[Примечание для студентов. Данный раздел устанавливает процесс внесения изменений в элементы конфигурации. Этот процесс должен быть достаточно гибким, чтобы изменения можно было делать быстро, и в то же время должен быть достаточно упорядоченным, чтобы отслеживать изменения и продвигать проект вперед без нанесения ущерба в результате изменений.]

3.2.1. Запрос на изменения

Как определено в SPMP (глава 2), команда назначает *инспектора* для каждого участника команды. Прежде чем сделать запрос на изменение, разработчик обязан

получить одобрение данного предложения по изменению от инспектирующей команды или, если последнее невозможно, от своего инспектора. Чтобы сделать запрос на изменение, необходимо предоставить форму http://www.ultracorp.division3.Encounter.submitCI ведущему конфигурацию и лидеру проекта вместе с исходным CI и измененным CI.

3.2.2. Оценка изменений

[Примечание для студентов. Для больших проектов группа людей, обычно называемая «группой управления изменениями», оценивает и одобряет (или отвергает) запросы на изменения. Студенческая группа должна сделать этот процесс по возможности простым.]

Лидер проекта или его заместитель оценивают все запросы на изменения. Лидер проекта должен также указать стандарты качества, которые необходимо учесть при внесении изменения.

3.2.3. Одобрение или неодобрение изменений

Запрос на изменение должен быть одобрен лидером проекта. Если лидер проекта не имеет возможности это сделать в течение трех рабочих дней, то право одобрения запроса на изменение переходит к ведущему конфигурацию.

3.2.4. Реализация изменений

[Примечание для студентов. Во избежание хаоса можно возложить на ведущего конфигурацию ответственность за реализацию изменений, но такое решение может породить узкое место на фазе реализации. Прежде чем возникнет «пробка», лидер проекта должен найти способ преодолеть данное узкое место, распределяя работу по реализации изменений между разработчиками настолько широко, насколько это возможно.]

Как только CI включается в конфигурацию, на ведущего конфигурацию возлагается ответственность за тестирование и интеграцию изменений. Это должно выполняться в соответствии с правилами регрессионного тестирования, описанного в документации по тестированию программного обеспечения (STD). В частности, ведущий конфигурацию должен координировать сборку версии для тестирования.

Выпуск версий должен утверждаться лидером проекта или руководителем проекта, если лидер отсутствует.

3.3. Определение статуса конфигурации

Ведущий конфигурацию обязан обновлять информацию о конфигурации на сайте http://www.ultracorp.division3/Encounter/Configuration не реже раза в неделю. Для этого достаточно опубликовать итоговый отчет инструмента SuperCMTool.

3.4. Аудиты и обзоры конфигурации

[Примечание для студентов. В промышленности часто применяют выборочные аудиты. Для студенческой команды это может быть слишком накладно ввиду недостатка ресурсов, но некоторые команды успешно справляются с этим. Периодические

обзоры, как часть регулярных собраний команды, требуют меньше времени, поэтому они рекомендуются к применению.]

Руководитель проекта должен запланировать выполнение обзоров конфигурации ведущим конфигурацию не реже, чем раз в две недели, обычно в качестве одного из пунктов повестки дня еженедельных собраний команды. Ведущий конфигурацию должен сделать обзор состояния конфигурации и предложить детальные процедуры управления конфигурациями на фазах кодирования и интеграции.

Вопросы управления конфигурациями должны быть предметом случайным образом назначаемых внешних аудитов.

3.5. Управление интерфейсом

Система управления конфигурациями должна иметь интерфейс с веб-сайтом проекта. Этим интерфейсом управляет ведущий конфигурацию.

3.6. Контроль поставщиков и субподрядчиков

Ведущий конфигурацию должен отслеживать обновления и сообщения об ошибках в инструменте SuperCMTool. У него должен быть план действий на случай, если поддержка SuperCMTool будет прекращена. Этот план должен быть представлен лидеру проекта в течение месяца после начала проекта.

4. Расписание

[Примечание для студентов. План мероприятий по управлению конфигурациями можно определить в данном разделе или в общем плане в SPMP. В последнем случае данный раздел должен не дублировать SPMP, а содержать только ссылку.]

План-график мероприятий по отчетам, архивации и обновлению конфигурации показан на рис. 1.22.

Рис. 1.22. Расписание управления конфигурациями

5. Ресурсы

Ведущему конфигурацию для выполнения своих обязанностей требуется в среднем приблизительно 6 часов в неделю в первой половине проекта и 12 часов в неделю во второй половине проекта. Время, затрачиваемое другими разработчиками на управление конфигурациями, принято отдельно не учитывать.

6. Сопровождение

[Примечание для студентов. Все документы проекта изменяются по ходу проекта, но SCMP особенно чувствителен к изменениям, поскольку в нем определяется способ управления изменениями.]

Ввиду важности наличия стабильного плана управления конфигурациями изменения в данном документе должны быть одобрены все командой проекта.

Поскольку целью организации является достижение уровня 5 по CMM, ведущий конфигурацию при подготовке совещаний по улучшению процесса управления конфигурациями обязан сделать следующее.

- Оценить эффективность данного плана.
- Количественно оценить потери, вызванные дефектами данного плана.
- Оценить эффективность использования инструмента SuperCMTool.
- Изучить литературу по новым методам управления конфигурациями; количественно оценить выгоды и затраты на проведение улучшений.
- Изучить новые инструменты управления конфигурациями.
- Предложить конкретные улучшения текущего процесса управления конфигурациями.
- Перечислить выгоды от улучшений.
- Предоставить оценки стоимости эффекта введения улучшений.
- Упорядочить предлагаемые улучшения по значению отношения выгоды–затраты.

Пример 2. План контроля качества (SQAP), часть 1

Утверждаю

_____Дата_____

Содержание данного SQAP следует стандарту IEEE 730-1989

17.01.99 Э. Брауде: Создание первой версии.

30.05.99 Р. Боствик: Рецензирование и добавление разделов от 7 до конца.

31.05.99 Э. Брауде: Интеграция и пересмотр содержания.

1. Цель

В этом документе описан план получения качественного продукта при выполнении проекта *Встреча*. Этот план включает средства обеспечения поддержки игры *Встреча*.

2. Задействованные документы

См. раздел 4.2.

3. Управление

3.1. Организация

[Примечание для студентов. Этот раздел устанавливает, какие роли задействованы в процессе обеспечения качества. Фактическое распределение обязанностей дано в разделе 3.3.]

Каждый участник команды отвечает за качество своей работы. Кроме того, на первые три итерации проекта *Встреча* назначается отдельный *ответственный за качество*. Ответственный за качество руководит всеми вопросами, связанными с обеспечением качества в проекте. Начиная с итерации 4 должна быть назначена команда инженеров по контролю качества, в которую должен войти ответственный за качество.

3.2. Задачи

[Примечание для студентов. Здесь определяется, что необходимо делать.]

Задачи по контролю качества включают в себя:

- документацию;
- обзорные собрания;
- верификацию (включая инспектирование);
- валидацию (прежде всего тестирование);
- виды деятельности, направленные на улучшение самого процесса обеспечения качества.

Эти задачи детализированы в данном документе.

3.3. Ответственность

[Примечание для студентов. Здесь определяется, кто что должен делать.]

В обязанности ответственного за качество входит следить за тем, чтобы задачи раздела 3.2 были сделаны, а требования данного документа были выполнены, в том числе чтобы были организованы обзорные собрания.

Лидер проекта отвечает за то, что управление качеством действительно производится.

[Примечание для студентов. Здесь не упоминаются конкретные имена, поскольку они должны быть указаны в подходящем месте, а именно в SPMP. Если продублировать здесь имена, то потребовалось бы обновлять более одного документа при изменении конкретных имен.]

Обязанности ответственного за требования и ответственного за проектирование описаны в разделе 6 данного документа.

4. Документация

4.1. Цель

В данном разделе определяется документация, используемая для обеспечения качества.

4.2. Минимальные требования к документации

[Примечание для студентов. В этом разделе перечисляются все документы проекта, поскольку именно проектная документация обеспечивает качество продукта.]

Должны быть созданы следующие документы.

- SQAP: План контроля качества (данный документ).
- SCMP: План управления конфигурациями.
- SPMP: План управления программным проектом.
- SRS: Спецификация требований к программному обеспечению.
- SDD: Проектная документация программного обеспечения.
- STD: Документация по тестированию программного обеспечения.
- Руководство пользователя.
- План сопровождения.

Помимо этих документов в исходном коде на Java будет использоваться Javadoc и, таким образом, можно будет генерировать документацию на уровне пакетов, классов и функций.

4.3. Прочее

План экспертизы программного обеспечения (SVVP) должен быть создан и поддерживаться независимо от SQAP.

5. Стандарты, практики, соглашения и метрики

5.1. Цель

Данный раздел описывает стандарты, практики, соглашения и метрики, используемые в проекте *Встреча*. Эти материалы призваны не только обеспечить качество проекта *Встреча*, но и получить количественные данные о самом процессе

контроля качества. Полученные данные должны быть использованы организацией Gaming Consolidated Industries (GCI) для повышения уровня по CMM с 3 до 4.

5.2. Содержание

[Примечание для студентов. Опишите стандарты, практики, соглашения и метрики, которые должны быть использованы. Здесь также можно указать общие цели организации в области качества или перенести их в специальное приложение. Содержание данного раздела должно быть конкретным. Например, следует избегать таких фраз, как «качество должно быть высоким, насколько это возможно».]

Стандарты.

Для ведения документации используются стандарты IEEE с соответствующими модификациями.

Практики.

1. Поскольку откладывать обеспечение качества накладно, компания GCI поощряет своих инженеров применять процедуры обеспечения качества непосредственно во время работы, а не откладывать на потом.
2. Компания GCI считает, что никакое средство обеспечения качества, внешнее по отношению к конкретному разработчику, не может само по себе обеспечить более высокое качество, чем это может сделать сам разработчик. Контроль качества в GCI рассматривается прежде всего как обучение, а не как система надзора или наказаний. Инженерам, которые еще не имеют опыта применения такой системы, рекомендуется войти в контакт с наставником по качеству.
3. Все артефакты проекта подлежат инспектированию и все они доступны команде после выпуска разработчиком. Это достигается путем помещения артефактов в систему управления конфигурациями, обеспечивающей доступ к содержимому в любое время.
4. Для всех процессов должен быть проведен обзор возможности улучшения хотя бы раз, и результаты этого обзора в письменной форме должны быть направлены в лабораторию технологии программирования (раздел 6.2.10).

Соглашения.

Правила и стиль написания документов должны следовать предложенным в книге [114], если это возможно. Служба контроля качества организует трехчасовые курсы по изучению принятых соглашений ежемесячно. Присутствие на этих курсах оплачивается из накладных расходов организации, а не из бюджета проекта.

Метрики.

Для каждого процесса и каждого документа должны измеряться по меньшей мере три показателя.

1. Время, затраченное сотрудником, на выполнение каждой подзадачи.
2. Самооценка качества по шкале от 1 до 10; эти данные не могут быть использованы руководством для оценки персонала, однако их отсутствие может быть принято во внимание.
3. Количество дефектов на единицу объема (например, на тысячу строк кода).

Компания GCI ставит перед собой цель достичь следующих показателей качества своих продуктов.

[Примечание для студентов. Числа, использованные ниже, должны опираться на собранные ранее данные.]

Ограничения на число дефектов, обнаруживаемых в течение двух месяцев после поставки.

- Требования: не более одного незначительного дефекта на сто детальных требований.
- Проектирование: не более одного незначительного дефекта на пять диаграмм.
- Псевдокод: не более двух незначительных дефектов на тысячу строк.
- Код: не более двух незначительных дефектов на тысячу строк кода, не считая комментариев.

Фактические данные по проекту должны быть представлены в приложении 1 к данному документу.

6. Обзоры и аудиты

6.1. Цель

Цель обзоров и аудитов состоит в том, чтобы привлечь внимание разработчиков к качеству приложения в ходе разработки. Обзоры проводятся на плановой регулярной основе. Объекты аудита выбираются случайным образом.

6.2. Минимальные требования

[Примечание для студентов. В больших проектах требуется полный список обзоров и аудитов, приведенный здесь. Студенческая команда как минимум должна провести обзоры и инспектирование требований и проектирования, а также заключительный обзор после окончания проекта. «Обзор» подразумевает обсуждение предлагаемого артефакта. «Инспектирование» применяется к законченному артефакту.]

План проведения обзоров описан в SPMP. Суть обзоров указана ниже.

6.2.1. Обзор требований к программному обеспечению

Данный обзор проводится по всему документу с требованиями в присутствии всей команды. Обзор ведет лидер проекта. Предполагается, что требования не были инспектированы до данного обзора. Данный обзор не подменяет инспектирование требований. Ответственный за требования обязан проследить, чтобы инспектирование требований состоялось (см. SPMP).

6.2.1A. Инспектирование требований к программному обеспечению

Все требования должны быть инспектированы в соответствии с Руководством по проведению инспектирования компании GCI, документ 345678.

6.2.2. Предварительный обзор проектных решений

Данный обзор альтернативных архитектурных решений выполняется всей командой. Обзор проводится лидером проекта или его заместителем. Подразумевается, что команда даст предложения, которые будут учтены в окончательном варианте проекта. Альтернативные архитектуры не должны инспектироваться до данного обзора. Ответственный за проектирование обязан проследить, чтобы данный обзор состоялся (см. SPMP).

6.2.3. Критический обзор проектных решений

Это инспектирование предложенной архитектуры в присутствии всей команды. Ответственный за проектирование обязан проследить, чтобы это инспектирование состоялось. Архитектура не должна инспектироваться до данного обзора. Если это возможно, архитектура должны быть разложена на составные части и по каждой из них должен быть проведен критический обзор проектных решений.

6.2.4. Обзор SVVP

Поскольку верификация и валидация должны выполняться независимой командой, в данном плане не предусматривается обзор SVVP.

6.2.5. Аудит функциональности

Лидер проекта перед поставкой продукта обязан организовать проверку того, что продукт соответствует SRS. Если необходимо сделать какие-то исключения, лидер проекта обязан получить предварительное разрешение на поставку от своего руководителя. Данные исключения должны быть доведены до сведения заказчика с помощью соответствующих средств, в том числе файла `README` и сопроводительного письма.

6.2.6. Аудит физических компонентов

До поставки ответственный за качество обязан организовать проверку того, что программное обеспечение и документация, предназначенные для поставки, физически включены в пакет поставки.

6.2.7. Аудиты процесса

Персонал проекта должны быть готов к внезапным аудитам своей работы. Аудит состоит в посещении рабочих мест командой, назначенной руководством отдела. Об аудитах следует предупреждать накануне. Предметом аудита является текущая работа отдельного разработчика или команды.

Поскольку организация движется в сторону CMM уровня 5, все рабочие материалы должны быть свободно доступны аудиторам и команде в любое время. Работа должна быть четко организована, так чтобы аудиты можно было проводить и без предупреждения.

6.2.8. Обзор управления

Высшее руководство должно проводить обзор проекта *Встреча* в первую неделю каждого месяца. Лидер проекта ответственен за организацию таких обзоров.

6.2.9. Обзор SCMP

Ответственный за качество должен проводить обзор состояния управления конфигурациями не реже раза в месяц независимо от процедур, установленных в плане управления конфигурациями.

6.2.10. Заключительный обзор

Команда проекта *Встреча* должна проводить заключительный обзор после выполнения каждой фазы, для того чтобы накопить данные для последующих проектов. В заключительный обзор включается как обзор только что завершенной фазы, так и обзор самого процесса контроля качества. Ответственный за качество обязан составлять отчет об улучшении процесса по каждой фазе и предоставлять его руководству лаборатории технологии программирования.

6.3. Инспектирование

Все артефакты проекта *Встреча* должны инспектироваться.

Разделы с 7 по 15 обсуждаются в главе 2.

Глава 2
Управление проектом

...О нас, о горсточке счастливцев, братьев.
Шекспир. Генрих V

Для успешной разработки приложений необходим подлинный командный дух. Повествуя о водопадном процессе разработки программного обеспечения, эта глава затрагивает разделы, выделенные на рис. 2.1.

- Основы: разделы 2.1–2.6
- Детали: разделы 2.7–2.15

Рис. 2.1. Схема разработки программ: темы главы

- Руководство по учебному проекту: План управления проектом для примера проекта *Встреча*.
- Упражнения.
- Примеры:
  1) План управления программным проектом (SPMP) для видеоигры *Встреча*.
  2) План контроля качества программного обеспечения (SQAP), часть 2.

Вопросы, рассматриваемые в этой главе.

- Понимание термина *управление проектом*.
- Организация команды.
- Выработка плана управления проектом.
- Определение рисков и их устранение.
- Оценка затрат на самых ранних стадиях жизненного цикла.
- Создание расписания проектов высокого уровня.
- Написание Плана управления программным проектом (SPMP).

Основы

2.1. Введение в управление проектом

2.1.1. Что такое управление проектом?

Управление проектом заключается в управлении производством продукта в рамках отведенных средств и времени. Поскольку для этого требуются человеческие ресурсы, то для управления проектом необходимы не только технические и организационные навыки, но еще и искусство управления людьми. Управление проектом далеко не обыденное дело. Оно может оказаться таким же захватывающим, как попытка посадить реактивный самолет на короткую взлетную полосу.

2.1.2. Составляющие управления проектом

Управление проектом охватывает:

- инфраструктуру (организационные моменты);
- управляющий процесс (права и ответственности участников);
- процесс разработки (методы, инструменты, языки, документация и поддержка);
- расписание (моменты времени, к которым должны быть представлены выполненные фрагменты работы).

2.1.3. Основные параметры: стоимость, функциональность, качество и расписание

Планировщики проекта могут варьировать стоимость, возможности, качество и дату завершения проекта. Руководитель проекта может управлять следующими факторами.

1. Общая стоимость проекта.

   Например, увеличивать расходы.

2. Возможности продукта.

   Например, удалять их из списка возможностей продукта.

3. Качество продукта.

   Например, увеличивать среднее время наработки на отказ (MTBF).

4. Длительность проекта.

   Например, сократить расписание на 20 % или отложить завершение проекта на один месяц.

Степень контроля над этими четырьмя факторами зависит от проекта. Несмотря на то, что *стоимость* может быть оговоренной заранее и фиксированной, зачастую существуют различные гибкие варианты. Предположим, что наш заказчик — химик, которому необходима визуализация молекул в 2D-графике. После просмотра прототипа молекулярной модели в трехмерной графике заказчик может положительно отнестись к дополнительным расходам, поскольку 3D-прототип выглядит намного лучше, чем двумерный. *Возможности* продукта также не являются фиксированными, как это может показаться сначала. Например, заказчик может отказаться от некоторого требования, если это сократит длительность проекта на 15 %. Хотя это может показаться ерундой, но даже показатели *качества* могут варьироваться. Если целевые значения показателей качества установлены слишком низко, это может привести к увеличению расходов на переделки из-за неудовлетворенности заказчика. Если же целевые значения показателей качества установлены слишком высоко, то затраты на поиски самой последней незначительной ошибки могут оказаться недопустимо высокими. Большинство заказчиков не пожелают платить тройную цену за текстовый процессор только ради того, чтобы получить версию, в которой устранен хорошо известный тривиальный дефект. *Дата* завершения проекта также может иногда сдвигаться. Например, руководитель может пожелать сдвинуть срок разработки, если продукт обладает столь многообещающими возможностями, что есть шанс завоевать рынок.

Один из способов визуализировать значения данных четырех переменных состоит в использования так называемых *лепестковых диаграмм*. В диаграммах данного типа значение каждой переменной откладывается на оси, исходящей из центра. Оси рисуются симметрично относительно центра. Конкретный пример лепестковой диаграммы показан на рис. 2.2. Поскольку в данном случае осей четыре, они перпендикулярны. На каждой оси центру соответствует наименее желательное значение показателя, а целевое значение откладывается на некотором расстоянии от центра. Таком образом, получается четырехугольник (если

бы показателей было пять, то получился бы пятиугольник, и т. д.). Например, на оси «Возможности» центр соответствует значению «0 % требований выполнено», а единица — «100 % требований выполнено». Фактические значения обычно находятся где-то в промежутке, хотя могут и выходить за пределы многоугольника, если удается перекрыть плановые значения показателей. Состояние проекта отражается путем закрашивания многоугольника фактических значений. Чем больше целевой многоугольник заполняется фактическим, тем точнее мы достигаем поставленных целей. Такая визуализация позволяет руководителю проекта правильно расставить приоритеты при принятии решений.

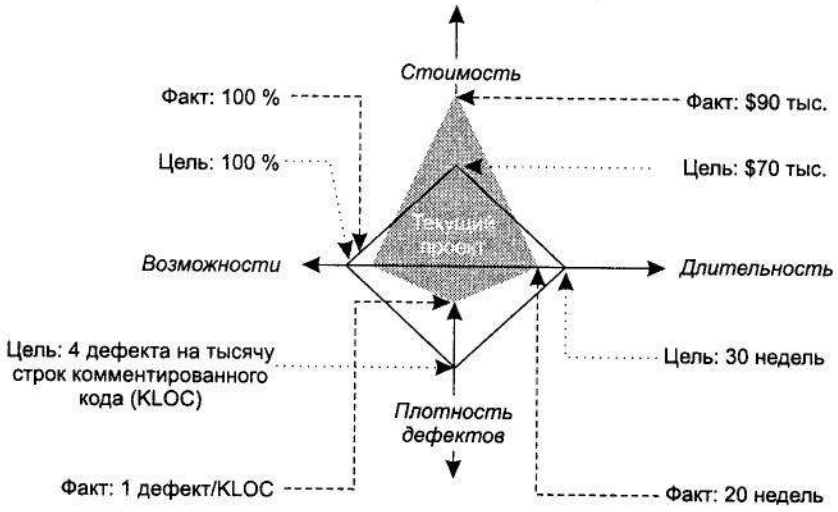


Рис. 2.2. Диаграмма для параметров проекта

Подводя итоги, можно констатировать, что управление проектом — это достижение баланса между стоимостью, возможностями, качеством и сроками. Профессиональный руководитель проекта оценивает этот баланс численно.

2.1.4. Типичная схема процесса управления проектом

Рассмотрим типичную последовательность действий, необходимую для запуска проекта.

1. Понять содержание проекта, область применения и временные рамки.
2. Определиться с процессом разработки (методы, инструменты, языки, документация и поддержка) (раздел 4 главы 1).
3. Выделить организационную структуру (привлечение отделов организации) (раздел 2.3).
4. Определить управляющий процесс (ответственность участников) (раздел 3 примера 1 в конце главы).

5. Разработать расписание проекта (моменты сдачи частей работы) (раздел 2.6).
6. Разработать план подбора кадров (раздел 3.5 примера 1 в конце главы).
7. Начать управление рисками (раздел 2.4).
8. Определить, какие документы необходимо выработать (раздел 4.2 SQAP).
9. Начать сам процесс. Описывается в главах 3–10.

К пониманию проекта (1) относится общее представление о целях и задачах, а не сбор всех требований. Последнее является длительным процессом, начинающимся после того, как набросан примерный вариант плана проекта (главы 3 и 4). Мы также должны понять общие возможности и назначение приложения. Банковское приложение может иметь как огромные возможности, например «Автоматизация всех банковских транзакций», так и совсем скромные, например «Сортировка по номеру чека», или нечто промежуточное. Необходимо прикинуть в общих чертах количество трудозатрат и времени на проект (например, два месяца или два года).

Далее (2) требуется принять решение об использовании той или иной модели процесса разработки (водопадной, спиральной, инкрементальной, комбинированной и т. д., раздел 1.4). Затем (3) следует провести общую организацию персонала (выяснить, какие отделы компании будут задействованы в проекте, установить количество команд, определиться с контролем качества). Как только все приготовления будут завершены, можно сосредоточиться на главной задаче для успешного выполнения работы — процессе управления (4). Здесь необходимо будет определить, кто перед кем отчитывается, или задать уровневую структуру типа TSP (представленную в главе 1 и рассматриваемую подробнее в этой главе). Следующий шаг (5) — это создание общего расписания (определяющего, когда что должно быть сделано). После этого можно завершать план подряда, в котором приводится количество персонала, привлекаемого к проекту, и примерное количество человеко-дней для различных задач (например, анализа требований). Этот план не может быть конкретным (например, кто над какой функцией будет работать), так как до сих пор еще не определена архитектура приложения. Проекты практически всегда подвергаются влиянию всяких неприятностей, поэтому определение проекта и управление рисками в проекте (7) необходимо начинать одновременно. Остается подготовить соответствующий набор документов (например, документы SVVP и SRS стандарта IEEE) (8), а затем запустить непосредственно сам проект (9).

2.2. Управление персоналом проекта

2.2.1. Профессионализм

Одной из тем, затрагиваемых в этой книге, является растущий уровень профессионализма в разработке программного обеспечения. Для того чтобы лучше понять, о чем идет речь, приведем пример. Обратимся к профессионализму хирур-

гов. Если хирург проводит операцию на головном мозге в среднем в течение четырех часов, то профессионализм не позволит ему подчиниться требованию администратора о сокращении времени операции до трех часов. Другими словами, *профессионалы* несут социальную ответственность, которая важнее удовлетворения нужд и требований их начальников и работодателей. С учетом того, что современный мир во многом зависит от программного обеспечения, программным разработчикам приходится брать на себя все большую ответственность. Это очевидно в случае создания автопилота на самолетах или космических кораблях, но это также верно и при автоматизированном расчете выплат по страховым полисам. Так же как люди не должны останавливаться перед каждым мостом, чтобы проверить его, прежде чем пройти, они совсем не обязаны проверять каждое вычисление на правильность.

Приобретение программными разработчиками навыков в планировании, составлении графиков работ, проектировании, реализации и тестировании артефактов проходит под тщательным как внутренним, так и внешним контролем. Интерес юристов к проблеме 2000 года является весомым доказательством заинтересованности общественности (то есть присутствием внешнего контроля). Применение метрик является составной частью решения проблемы. Такие авторы, как Хэмфри [55], советуют программным разработчикам не обещать заранее точной даты завершения проекта, а периодически проводить переговоры. В прошлом для разработчиков было проблематично более или менее точно рассчитать для проекта время и трудозатраты. Зачастую для этого просто не хватало инструментальных средств. По мнению автора, ситуация должна измениться. Командный процесс разработки программного обеспечения (TSP), предложенный Хэмфри, является одним из примеров достаточно конкретной процедуры для производства с использованием согласованного набора метрик.

С 1994 по 2000 год объединенная комиссия IEEE и ACM работала над определением критериев, по которым можно выделить *профессиональных* разработчиков программного обеспечения.

2.2.2. Важность управления персоналом

Одной из главных составляющих, необходимых для производства программного обеспечения, являются люди. Несомненно, в первую очередь оцениваются технические навыки инженеров. Однако эти навыки необходимо применять для решения проблем в нужное время и в нужном месте. Это предполагает комбинацию работы в команде и лидерства. Нашей целью в этом разделе является знакомство с типичными взаимосвязями в коллективе. Мы рассмотрим управление персоналом проекта с различных точек зрения, а именно с точки зрения предприятия, которому необходимо приложение, менеджеров, ответственных за управление созданием этого приложения, и с точки зрения задействованных в проекте разработчиков.

Брукс стал знаменитым благодаря своей книге «Мифический человеко-месяц» [18], которая содержит «Закон Брукса». В соответствии с этим законом привлечение большего числа людей в погибающий программный проект не помогает, а может даже и навредить. Другими словами, дополнительные челове-

ко-месяцы, неожиданно появляющиеся в ходе проекта, на самом деле являются мифом. Однако при хорошем управлении дополнительные человеко-месяцы могут оказаться очень полезными.

Книга «Peopleware», написанная Де Марко и Листером [24], является еще одной влиятельной работой в области управления проектами. В ней авторы утверждают, что если бы они могли управлять одними и теми же программными проектами снова и снова (что неосуществимо), то делали бы это каждый раз по-другому. Книга «Peopleware» состоит из коротких эссе, в каждом из которых рассматривается одна из «проторенных дорожек», по которым проходят менеджеры, часто сожалея, что они выбрали этот путь. Обычно к ошибкам в управлении людьми ведет бездумное следование устоявшимся стереотипам и предрассудкам, которые всем известны, затвержены в фольклоре, но зачастую ложны. Например, «Возьмите оценку времени, которую назвал сотрудник, и удвойте ее», «Завинчивайте гайки», «Не позволяйте работать дома»... По наблюдениям автора, все без исключения разработчики хотят, чтобы с ними обращались, как с профессионалами, и делают все возможное, чтобы получить профессиональные результаты.

2.2.3. Корпоративные аспекты

С точки зрения предприятия проведение проекта является вкладом в достижение общих целей предприятия, то есть производство продукта, который приносит прибыль. Это чисто деловой подход.

2.2.4. Управленческие аспекты

Говорят, что «не бывает технических провалов, а бывают только провалы в управлении». Хотя это и не совсем правильно, однако напоминает нам о важности управления для успешного завершения деятельности по разработке. Отношение менеджера к своим подчиненным — это смесь делового интереса с интересом работы с людьми, участвующими в проекте. Несмотря на частые опасения коллектива, среднестатистический менеджер старается довести работу до конца так, чтобы при этом у подчиненных не возникало чувство недовольства или неудовлетворенности. Менеджер сам в этом заинтересован, так как недовольные сотрудники работают непродуктивно. Одной из самых трудных задач менеджера является достижение главной цели — выполнения работы при условии получения коллективом удовлетворения от проделанной работы. Например, высшее руководство может поставить условие, что работа должна быть выполнена на старом языке программирования для совместимости версий, хотя программные разработчики хотели бы опробовать новый язык или систему.

Менеджер должен сделать так, чтобы технические изыскания разработчиков имели нужное направление. Начинающие менеджеры зачастую находят довольно сложным лавирование между отданием приказаний и предоставлением свободы разработчикам. Диктатура в управлении может вызвать негодование коллектива и привести к потере мотивации. Однако излишнее попустительство может привести к потере времени и напрасно проделанной работе. Решением данной про-

блемы является лидерство, выяснение истинных желаний и потребностей людей, активное их распределение и объединение в попытке достижения успеха. Лидеры проекта должны варьировать степень своей ответственности в управлении, основываясь на величине проекта. В больших проектах их обязанности состоят в основном в управлении. В малых же проектах лидеры должны обеспечивать как общее управление, так и личное участие непосредственно в разработке.

2.2.4.1. Организация совещаний

Одной из обязанностей руководителя проекта является проведение совещаний. Поскольку команды, как правило, не очень хорошо могут создавать артефакты «с ходу» (особенно это касается проектирования), очень полезно иметь заготовку («рыбу») в качестве основы дискуссии. Например, ответственный за проектирование может подготовить эскиз архитектуры, или руководитель проекта может подготовить предварительное распределение работы. Эти эскизные проекты не должны быть очень конкретны, чтобы оставалась возможность для творческого вклада участников команды.

ОДИН ИЗ СПОСОБОВ ПЛАНИРОВАНИЯ И ПРОВЕДЕНИЯ СОВЕЩАНИЯ

Эти рекомендации по проведению совещаний могут быть полезны всем участникам команды. Первые два пункта должны выполняться заблаговременно до начала совещания.

1. Прикиньте приблизительно время начала, время окончания и повестку совещания.
   - Сначала перечислите самые важные вопросы.
2. Подготовьте «рыбу» решений.
3. Начинайте вовремя.
4. Назначьте кого-либо фиксировать ход совещания (это должны обеспечить участники совещания).
5. Назначьте кого-нибудь следить за временем и подсказывать имена участников совещания.
6. Добейтесь согласия по повестке совещания и по времени его проведения.
7. Следите за временем и завершайте совещание вовремя:
   - допускаются исключения в случае важной дискуссии;
   - прекращайте лишние дискуссии, делайте перерывы.
8. Поддерживайте разговор по существу вопроса.
9. После совещания высылайте электронной почтой основные моменты, принятые на совещании решения и его итоги.

Многие участники совещаний, особенно студенты, считают, что совещания продолжаются слишком долго, не принося видимых результатов. Но если примерное время дискуссии согласовано и ограничено заранее, то участники обычно более сфокусированы на обсуждаемом вопросе и совещания становятся более эффективными. Руководитель проекта обязан решить, стоит ли продолжать дискуссию, или нужно ее прекратить. Ключевыми факторами являются следующие: является ли дискуссия продуктивной и не отнимает ли она времени у дискуссии по еще более важному вопросу. Руководитель обязан следить, чтобы дискуссия не отклонялась от темы и вела к каким-то выводам. Время от времени руководитель обязан вмешаться и принять волевое решение, потому что консенсус не всегда возможен. Секретарь, фиксирующий ход совещания, должен записывать

принимаемые решения. Это нужно делать в форме итогов, не углубляясь в детали. Детали отражаются в проектной документации.

Хорошей практикой является предварительная подготовка повестки дня совещания.

ОДИН ИЗ СПОСОБОВ СОЗДАНИЯ ПОВЕСТКИ СОВЕЩАНИЯ

1. Получите согласие по повестке и времени проведения совещания.
2. Найдите добровольцев для:
   - записи принимаемых решений и обсуждаемых вопросов;
   - слежения за временем и напоминания имен участников совещания.
3. Отчет о прогрессе проекта в соответствии с расписанием — 10 минут.
4. Обсудите заготовки артефактов — x минут.
5. Обсудите меры по снижению риска — 10 минут:
   - другие вопросы;
   - имеет ли место улучшение метрик и процесса?

n. Обзор намеченных действий — 5 минут.

В приведенном примере повестки совещания упомянуты метрики и улучшение процесса: это полезно обсуждать по завершении фазы, но отнюдь не на каждом совещании. Меры по снижению риска (раздел 2.4) нужно ставить в повестку каждого совещания на первой трети проекта. В дальнейшем идентификацию и снижение рисков также стоит обсуждать, но не так часто.

2.2.5. Человеческий фактор

Наконец, рассмотрим *человеческие факторы*, характерные для типичного разработчика программного обеспечения. Разработчики хотят иметь интересную работу, хотят иметь возможность проявить себя, хотят, чтобы их заметили и наградили, и хотят теплых дружеских отношений в коллективе. Здоровое самоуважение является предпосылкой этих желаний. Одним из источников самоуважения является качественная работа. В этих наблюдениях нет ничего нового. Если вы наняли плотника построить кухню и показываете, что цените умелую работу, то плотник, скорее всего, будет мотивирован. Если вы ничего не понимаете в плотницком мастерстве, то менее вероятно, что плотнику понравится задание. Следовательно, необходимо, чтобы разработчики точно знали, что означает *качество*. Например, разработчики должны знать, как оценить трудозатраты, необходимые для производства хорошего продукта, как доказать себе и другим, что работа сделана корректно, и как измерить качество сделанной работы.

2.3. Варианты организации персонала

Мы обсудим решение организационных вопросов в команде проекта с двух точек зрения. Первая – задание структуры несения ответственности. Вторая заключается в подборе кадров и создании системы отчетности подчиненных перед

начальством. В любом случае трудно чего-либо добиться при отсутствии налаженных взаимоотношений в коллективе.

2.3.1. Управление взаимодействием

Личный опыт автора, да и многих других, показывает, что количество разработчиков, с которыми каждому из разработчиков необходимо регулярно общаться, составляет от трех до семи человек. (Хэмфри [55] предлагает от четырех до восьми человек.) Результаты различных опытов по определению количества членов команды для наиболее эффективной работы сильно разнятся. Экстремальные значения, которые могут быть полезны при определении численности команды, приведены на рис. 2.3. С одной стороны, разработчик может работать без регулярного индивидуального общения с кем бы то ни было. Однако такая изоляция обычно приводит к непониманию разработчиком предъявляемых к нему требований и, как следствие, к относительно низкому уровню эффективности. С другой стороны, разработчик может работать в такой большой команде, что из-за постоянного общения с каждым из коллег у него не останется времени на собственно разработку. Это снова приводит к относительно низкому уровню эффективности. В частности, *регулярное общение* означает разговор с кем-либо в течение примерно двух часов в неделю. Таким образом, если инженер регулярно общается с десятью своими коллегами, то добрая половина его рабочего времени проходит в разговорах. Организаторы проектов, планирующие проекты с командами в двадцать, а то и в сто человек, обязательно должны учитывать все вышенаписанное.

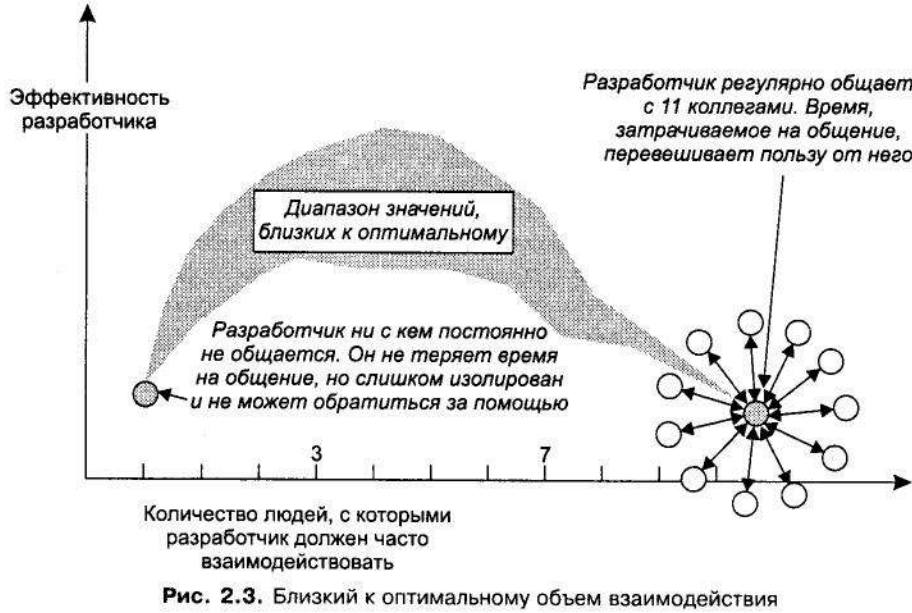


Рис. 2.3. Близкий к оптимальному объем взаимодействия

2.3.2. Варианты структуры ответственности

Иерархическая структура управления (рис. 2.4) является одним из предельных вариантов структуры управления. В этой организации существует главный управляющий — это Эйприл Смит, с тремя подчиненными управляющими. Лайл Герберт несет ответственность за аспекты проекта, связанные с маркетингом. Очевидно, что отдел маркетинга будет связываться с заказчиками для поддержания уверенности в том, что продукт удовлетворяет заказчика. Куин Паркер и его подчиненный Верн Крупп отвечают за контроль качества. Достоинства этой организационной схемы заключаются в том, что каждый понимает линии руководства и пути принятия решений, к тому же количество людей, с которыми каждый из сотрудников должен регулярно общаться, является удовлетворительным. Недостатком является то, что члены команды не участвуют в постановке задач, которые в основном ставятся перед ними начальством. При прочих равных обстоятельствах это один из наиболее безопасных путей организации проекта. Крупные проекты, организованные в таком иерархическом стиле, требуют проработки более широких и глубоких организационных схем.

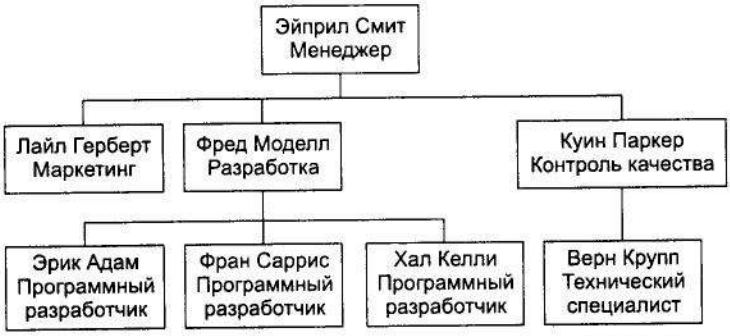


Рис. 2.4. Иерархическая организация управления проектом

Другим предельным вариантом структуры управления является *команда равных*, состоящая из сообщества сотрудников с одинаковыми правами и возможностями. Достоинством такой организации является огромный потенциал мотивации, который появляется благодаря равноправному партнерству в проекте. Зачастую эта структура прекрасно работает, если рабочая группа — небольшая, достаточно компетентная и сработанная. К недостаткам относятся трудности при решении разногласий и тот факт, что «никто ни за что не отвечает». По личному опыту автора, простое постановление о коллегиальности и консенсусе в принятии решений не означает, что процесс автоматически начнет лучше работать. Командный процесс разработки программного обеспечения, изобретенный Хэмфри [51], — это своеобразный набор путеводных нитей для таких команд. В конечном счете должна быть получена смесь коллегиального участия и личной ответственности в соответствии с размерами проекта, его природой, зрелостью и привлекаемыми к проекту людьми.

Один из примеров горизонтальной структуры управления приведен на рис. 2.5.

Рис. 2.5. Горизонтальная организация управления проектом

Идея состоит в том, что участники команды равны, за исключением Гила Варнера, который является ответственным за проектирование. В идеале он поощряет участие всех разработчиков в проектировании, но если требуется, принимает решения единолично. Более детально возможная организация команды показана ниже. Если в команде пять участников, то один и тот же участник может быть ответственным за требования и по реализации, поскольку в каждый конкретный момент времени только одно из двух требует повышенной активности. Ответственность за требования можно распределить между всеми, но кто-то конкретно должен отвечать за SRS. Можно меняться ролями примерно раз в три месяца с целью приобретения опыта. Поскольку каждая роль критична, разумно предусмотреть *систему подстраховки*. Страхующий инженер берет обязанности на себя, если назначенный лидер не может их выполнить по каким-то причинам. Страхующий обязательно инспектирует всю работу страхуемого, чтобы быть в курсе дела.

ОДИН ИЗ СПОСОБОВ ОРГАНИЗОВАТЬ КОМАНДУ

Схема организации команды с применением взаимного страхования имеет следующий вид.

1. Выберите лидера команды, который будет выполнять следующие обязанности:
   - обеспечивать активность всех аспектов проекта;
   - восполнять все пробелы.
3. Определите основные роли и задокументируйте ответственность:
   - ↑ лидер команды: предлагает и поддерживает... SPMP;
   - ↑ ответственный за конфигурацию:... SCMP;
   - ↑ ответственный за качество:... SQAP, STP;
   - ↑ отвественный за требования:... SRS;
   - ↑ ответственный за проектирование:... SDD;
   - ↑ ответственный за:... программный код.
2. Обязанности лидера:
   - предложить заготовку артефакта (например, SRS, SDD);
   - добиться улучшения и принятия командой;
   - обеспечить техническую поддержку и инспектирование артефакта;
   - поддерживать соответствующие метрики, если нужно.
4. Определите страхующего для каждого из ответственных, как это показано стрелками выше.

Такая схема обеспечивает гладкую передачу артефактов из рук в руки при смене фаз, поскольку каждый заранее знакомится с артефактом, за который ему предстоит отвечать на следующей фазе.

С ростом числа участников проекта организация команды равных становится невозможной, поскольку количество связей по взаимодействию растет квадратично. Действительно, между тремя участниками есть три связи, четыре участника имеют шесть связей, пять человек — десять связей, то есть n человек имеют $(n-1)+(n-2)+...+1=n(n-1)/2$ связей (каждый с каждым). Таким образом, 100 человек должны участвовать в 4950 взаимодействиях! Для больших проектов альтернативой является схема организации, приведенная на рис. 2.6. В этой схеме команды равных небольшие, но один участник команды выделен для взаимодействия с другими командами. Такая схема сохраняет преимущества небольших команд, но позволяет большому количеству людей создавать большие приложения.

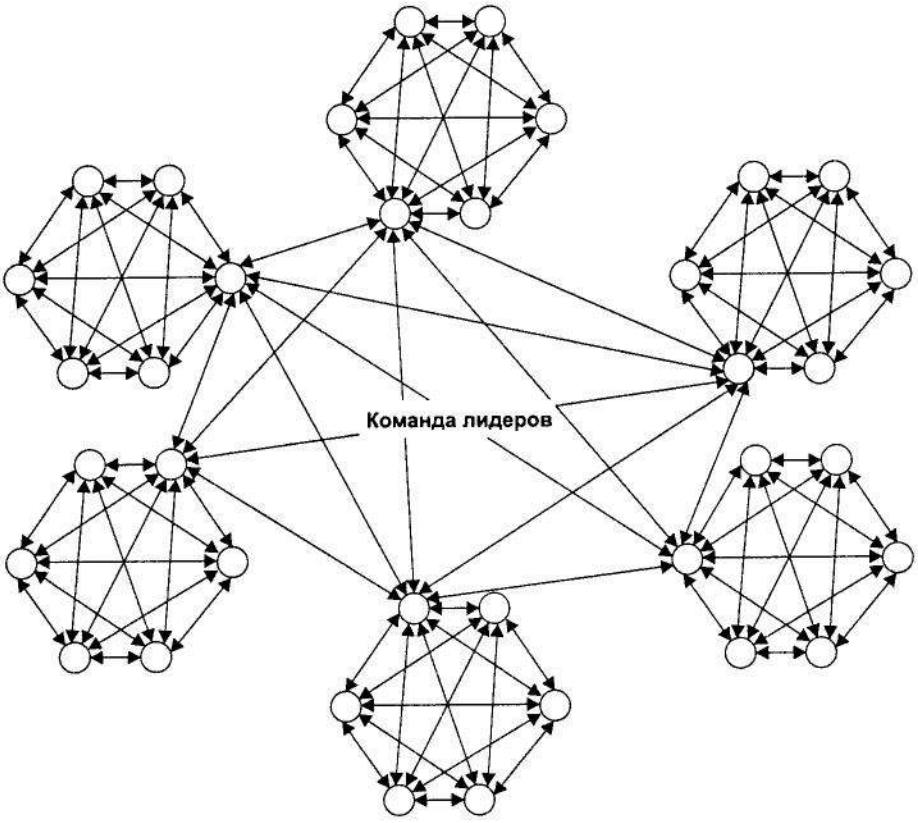


Рис. 2.6. Коллегиальная организация для сравнительно больших проектов

Существует расхожее мнение, что хороший инженер и хороший менеджер редко совмещаются в одном человеке. Однако автору случалось наблюдать многих инженеров, которые были отличными лидерами групп, даже если мало кто ожидал этого от них.

2.3.3. Подбор участников проекта

В *проектно-ориентированных* организациях персонал проекта организационно отчитывается перед руководителем проекта. Менеджер является начальником персонала. Программный разработчик всегда прикреплен к какому-либо конкретному проекту и организационно не связан с другими разработчиками в других проектах внутри компании. Преимуществом является упрощение вертикали власти, однако основной недостаток заключается в профессиональной изоляции разработчиков. Например, очень маленькие компании почти всегда организованы по принципу проектной ориентации, однако даже огромные компании иногда организуются так же, особенно когда хотят произвести впечатление на заказчика.

Вообще говоря, системы отчетности на практике имеют более сложную и комплексную структуру, чем в проектно-ориентированных организациях. Это происходит из-за использования *матричной структуры*. В матричных организациях служащие относятся к функциональным подразделениям (например, отдел разработки, отдел продаж и т. п.) и выделяются этими подразделениями для участия в том или ином проекте (табл. 2.1). Таким образом, начальник программного разработчика, отвечающий за своего подчиненного, будет членом функционального подразделения программной разработки. Однако в каждом проекте, в котором разработчик принимает участие, он подчиняется лидеру проекта. Обычно разработчиков на постоянной основе привлекают не более чем к двум проектам. Например, если бы некая организация, использующая иерархическую организацию проекта (см. рис. 2.4), имела матричную структуру, то Лайл Герберт был бы членом отдела маркетинга. При этом его начальником был бы не Эйприл Смит, а управляющий отдела маркетинга, который консультировался бы с Эйприлом при оценке работы своих подчиненных. Основным достоинством матричных организаций является профессионализм и возможность улучшения навыков. К недостаткам относится нечеткость в линиях субординации. Довольно часто компании стараются найти золотую середину между матричной и проектно-ориентированной структурами.

Таблица 2.1. Матричная организация

Отдел	**Проект**			
	Проект бронирования билетов на авиалинии	**Проект системы банковских расчетов**	**Проект молекулярного анализа**	**Проект механики текучих сред**
Отдел управления проектом	Эл Пруит	Куин Паркер	Рут Пелла	Фред Пасонс
	Полная загруженность	Частичная загруженность	Полная загруженность	Полная загруженность
Отдел маркетинга	Оскар Мат	Пит Меррилл	Сью Мо	Элтон Мастон
	Полная загруженность	Полная загруженность	Частичная загруженность	Полная загруженность
Отдел разработки	Хал Эгбертс	Бен Эрлих	Мэри Эриксон	Лен Ингелс
	...	...	...	...

2.4. Выявление и уменьшение рисков

В этом разделе мы ненадолго станем людьми с повышенной тревожностью. Мы будем думать, что наш проект на грани срыва из-за действия неких сил. Затем мы воздвигнем мощную защиту.

2.4.1. Что такое риски

Риск — это нечто, что может появиться по ходу проекта, и это нечто в худшем случае может негативно повлиять на проект. Промышленные данные демонстрируют огромное количество сорванных программных проектов. Например, Rational Corporation приводит следующий факт: «более 70 % всех программных проектов находятся под угрозой или испытывают серьезные трудности». Факторы, приводящие проект к срыву, на ранних стадиях проекта проявляются в виде рисков. Таким образом, своевременное выявление того или иного риска, а также принятие соответствующих мер позволяют предотвратить срыв проекта. Существуют два типа рисков.

1. Риски, которых можно избежать или которые можно предотвратить (устранимые).
2. Риски, которых невозможно избежать.

Примером первого типа рисков является следующий: «Что если руководитель проекта в команде из 15 человек уходит из компании?» (риск устраняется подготовкой заместителя руководителя проекта). Пример риска второго типа: «Необходимо собрать данные о 2100 полетах путем опроса персонала аэропорта, прежде чем мы сможем поставить продукт».

Если риск первого рода выявлен достаточно рано, то меры по предотвращению риска позволяют успешно продолжать проект. Очень полезно выявлять и риски второго рода. Можно либо сразу прекратить проект и избежать дальнейших потерь, либо изменить область применения или иным образом изменить сам проект, чтобы минимизировать риск.

Эффективные команды принимают наличие рисков как данность и постоянно о них думают.

2.4.2. Обзор управления рисками

Приложения, похожие на те, которые были разработаны ранее теми же инженерами, как правило, не являются рискованными. Однако разнообразие программных приложений огромно и быстро растет. Многие работы содержат новые пути представления задач или являются реализациями новых идей. По этим причинам в разработке программных приложений обычно присутствует много рисков.

Каждый идентифицированный риск должен с радостью восприниматься командой проекта, так как в этом случае с ним можно начать что-то делать. *Настоящей проблемой являются риски, которые не удалось идентифицировать.* Такие риски похожи на мины, ждущие свои цели. Поскольку процент проектов, которые уже

никогда не завершатся, поистине огромен, то постоянное внимание к рискам делает более вероятным тот факт, что проект попадет в 20 % более удачливых проектов или будет прекращен до того, как будут потрачены громадные деньги и загублены карьеры инженеров.

Управление риском состоит из нескольких действий.

1. Идентификация.

   Старайтесь постоянно обнаруживать риски.
2. Планирование устранения.
3. Выбор приоритетов.
4. Устранение или уменьшение.

Эти шаги требуется осуществлять с самого начала проекта и упорно продолжать в течение первой четверти проекта. Некоторые команды назначают одного из своих коллег на роль координатора рисков, ответственного за поиск рисков, а также информирующего об их устранении.

2.4.3. Выявление рисков

Идентификация риска заключается в фиксации всех факторов беспокойства и озабоченности, связанных с проектом, а затем в постоянном обдумывании всей командой других возможных опасений. Для обнаружения рисков необходим скептический склад ума. Идентификация риска сродни проведению инспектирований — глобальному поиску дефектов в плане разработки. Различают несколько категорий риска: переоценивание рабочего времени, слишком быстрые изменения требований, неготовность найти достаточно эффективный инструментарий, недостаточные навыки персонала, нехватка времени для обучения работы с новыми инструментами (например, CASE-инструментами), несовершенство языков (например, слишком медленная работа приложения). Факторы риска для самого простого проекта перечислены ниже. Они были предложены Кейлом и др. [73] в ходе исследований в США, Гонконге и Финляндии.

1. Недостаточная вовлеченность в проект высшего руководства.
2. Невозможность привлечения пользователей.
3. Непонимание требований.
4. Привлечение неадекватных пользователей.
5. Невозможность управления ожиданиями конечных пользователей.
6. Изменение области применения или целей проекта.
7. Нехватка знаний или навыков у персонала.

Может показаться странным, что первые два риска связаны с обязательствами вкладчиков — людей, наиболее заинтересованных в выпуске приложения. (Вкладчик — это некто, претендующий на прибыль от проекта.) Причем количество вкладчиков может быть достаточно большим. Как и в большинстве кол-

лективов, у каждого члена общества вкладчиков свои мотивации, которые порой сложно согласовать с мотивациями других людей. Ошибки при согласовании мотиваций приводят к нестабильным требованиям, которые могут разорить проект. Заметим, что большинство категорий из этого списка имеют ту же природу, что и источники риска № 1 и № 2, за исключением пункта 7. Заслуживает внимания тот факт, что технические моменты составляют всего 20 % от десяти наиболее важных факторов и рассматриваются как менее важные по сравнению с другими. Остальные факторы можно отнести к политическим и организационным. Все это можно подытожить, сказав, что львиная доля работы по предотвращению неприятностей на пути к успешному завершению проекта ложится на плечи руководителя проекта.

2.4.4. Предупреждение рисков

Предупреждение рисков — это процесс, в ходе которого степень рисков снижается или риски полностью устраняются. Имеются два способа предупреждения рисков. Первый заключается во внесении изменений в требования проекта, благодаря чему устраняется причина возникновения риска (*избежание риска*). Другой способ — разработка неких технологий и архитектуры, решающих проблему (*преодоление риска* или, проще говоря, его устранение). Предупреждение рисков можно сравнить с прокладыванием маршрута пешехода в обход препятствий — улицы и дома, как показано на рис. 2.7.

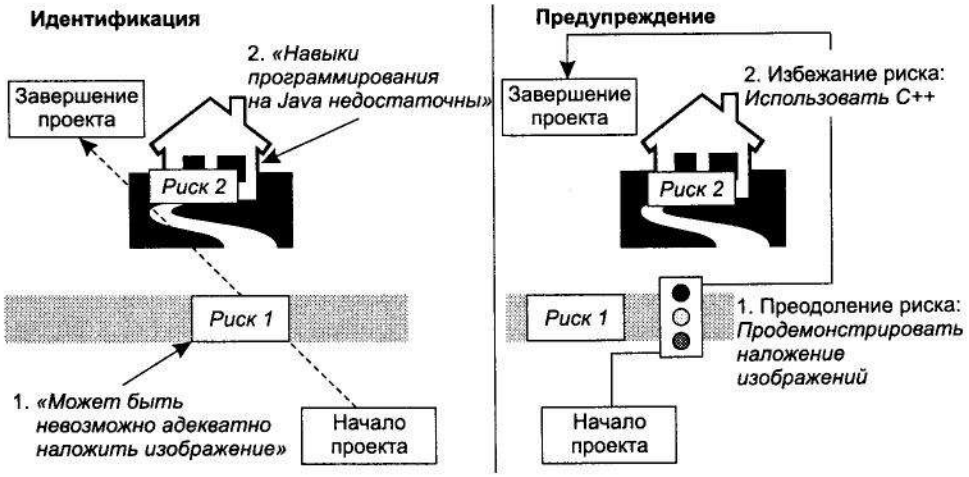


Рис. 2.7. Приемы управления рисками

Как только команда обнаруживает некое препятствие на маршруте следования, она может либо обойти его, скорректировав направление движения, либо немедленно начать работу по его преодолению. Другими словами, риски можно либо избежать (на рис. 2.7 поиск пути вокруг дома, а не через него), либо преодо-

леть (установка светофора для безопасного перехода через улицу). Два риска из проекта *Встреча*, приведенные на рис. 2.7, так же как и их предупреждение, поясняются ниже.

В хорошем проекте риски постоянно идентифицируются и обычно имеется список рисков, ждущих своей очереди для обработки. В этом списке риски должны быть упорядочены по приоритетам, поскольку обычно нет времени проводить мероприятия по устранению для всех рисков. В хорошо управляемом проекте те риски, которые не были предупреждены заранее, являются наименее опасными и даже если случаются, то не создают серьезных проблем. Схема расстановки приоритетов объясняется в табл. 2.2. Для каждого риска определяются три величины: вероятность осуществления риска; ущерб, наносимый проекту данным риском в случае осуществления; оценка стоимости устранения риска. Для всех величин используется одна шкала, например от 1 до 10. Первые два числа вычитаются из 11, третье берется как есть и все полученные числа перемножаются, а результат считается приоритетом риска.

Таблица 2.2. Метод расчета приоритета рисков

	Вероятность 1–10 (1 = маловероятно)	**Ущерб 1–10 (1 = слабое влияние)**	**Стоимость устранения 1–10 (1 = низкая стоимость устранения)**	**Расчет приоритета**	**Итоговый приоритет Чем меньше число, тем выше приоритет**
Высокоприоритетный риск	10 (большая вероятность)	Сильное влияние	1 (низкая стоимость устранения)	(11–10)× ×(11–10)×1	1
Низкоприоритетный риск	1 (маловероятно)	1 (слабое влияние)	10 (высокая стоимость устранения)	(11–1)× ×(11–1)×10	1000

Два риска для видеоигры *Встреча* идентифицированы в табл. 2.3. Риск № 1, *наложение изображений*, связан с манипулированием изображениями в Java. Предположим, что никто в команде не пробовал накладывать изображения. Однако это необходимо делать, поскольку изображения персонажей должны двигаться на фоне изображения зоны. Расчет приоритета и план предупреждения риска также указаны в табл. 2.3. Мы предполагаем, что данный риск действительно может стать проблемой, но более вероятно, что все обойдется, поэтому мы ставим вероятность 3. Но если это случится, то проблема будет весьма серьезной, поэтому ущерб мы оцениваем максимальным числом 10. Мы предполагаем, что стоимость предупреждения данного риска мала и оцениваем ее числом 1. План предупреждения очень прост: нужно поручить одному из инженеров изучить документацию и на практике проверить, что при наложении изображения персонажа поверх фона все происходит нормально (в частности, не видно прямоугольной границы накладываемого образа).

Таблица 2.3. Образец анализа рисков для игры Встреча

Номер риска	Название риска (детали оговорены выше)	Вероятность 1–10 (1 = маловероятно)	Ущерб 1–10 (1 = слабое влияние)	Стоимость устранения 1–10 (1 = минимальная стоимость устранения)	Приоритет Чем меньше число, тем выше приоритет	План устранения (уменьшения)	Ответственный разработчик	Дата завершения работ
1	Наложение изображений	3	10	1	8	Экспериментирование с наложением изображений в Java	П. Р.	02.01.99
2	Недостаточные навыки программирования на Java	9	6	8	10	Посещение Х. Т., К. М., В. И. и Л. Д. учебных курсов, начинающихся 05.01.99 в Ultra Training Corp. Получение сертификатов Java: уровень 2 — 01.03.99; уровень 3 — 15.04.99	Х. Л.	04.15.99
3	Алан Грэй может быть снят с этого проекта	3	7	9	288	Инспектирование Сьюзан Феррис всей работы, которую делает Алан Грэй	С. Ф.	Непрерывно

Риск № 2, *недостаточные навыки программирования на Java*, отражает тот факт, что 40 % команды не имеют достаточного опыта программирования на Java. Конечно, это вопрос оценки умений, но предложивший данный риск абсолютно уверен, поэтому вероятность равна 9. Принимая этот факт, следует признать, что данный фактор серьезно затормозит проект, но поскольку не все в команде должны программировать, ущерб оценивается числом 6. Стоимость устранения риска высока, поскольку нужно затратить много времени на обучение с отвлечением от работы, так что полагаем ее равной 8.

Использование подобных метрик весьма полезно, но всегда должно критически оцениваться с позиций здравого смысла. Например, имеет смысл отдельно рассматривать риски, которые имеют высокую вероятность и осуществление которых может остановить проект. Например, хотя риск № 1 имеет более высокий приоритет, риск № 2 требует больше времени на устранение, так что его устранение нужно начинать немедленно. Команды должны рассматривать риски с различных точек зрения. Если есть много серьезных рисков, может быть лучше отложить начало проекта до тех пор, пока они не будут устранены.

ОДИН ИЗ СПОСОБОВ ОПРЕДЕЛЕНИЯ И ПРЕДУПРЕЖДЕНИЯ РИСКОВ

Описываемый процесс нацелен на наиболее продуктивное использование времени совещаний за счет автономного выполнения работы, которая не требует присутствия всей команды. Первые три пункта выполняются заранее, до первого совещания.

1. Каждый член команды затрачивает 10 минут на выявление основных опасений по поводу успешного окончания проекта.
2. Каждый участник специфицирует эти риски в конкретных терминах, оценивает их, предлагает планы устранения (формат см. выше) и отсылает все это лидеру команды по электронной почте.
3. Лидер команды суммирует результаты и расставляет приоритеты.
4. Команда затрачивает 10 минут на поиск дополнительных рисков (на совещании).
5. Команда затрачивает 10 минут на окончательное формирование таблицы рисков (на совещании).
   - Назначаются разработчики, ответственные за устранение рисков.
6. Разработчики устраняют риски (между совещаниями).
7. Команда на еженедельных совещаниях в течение 10 минут делает обзор рисков:
   - ответственные за устранение рисков разработчики докладывают о прогрессе;
   - команда обсуждает новые обнаруженные риски и добавляет их в таблицу рисков.

2.5. Инструментальные средства разработки и поддержки

2.5.1. Модели процесса

Необходимо принять решение об использовании той или иной модели процесса разработки или комбинации моделей. На выбор представлены: водопадная, спиральная, унифицированная и инкрементальная модели, рассмотренные в главе 1.

2.5.2. Инструментальные средства

Разработка программного обеспечения представляет собой объемный рынок, на который также поставляются инструментальные средства в помощь инженерам, занимающимся производством программных приложений. Часто этот инструментарий называют *инструментарием для автоматизированной разработки программ* (CASE — Computer-aided Software Engineering). В течение многих лет обсуждается, что же относится к CASE-инструменту. В свое время разработчиками CASE-инструмента было многое обещано, однако сделано было намного меньше. Компоненты CASE-инструмента могут быть, например, такими.

- Для обеспечения управления проектом:
  - работа с расписанием;
  - распределение работ.
- Для обеспечения управления конфигурациями.
- Для управления требованиями.
- Для проектирования:
  - функциональные;
  - объектно-ориентированные;
  - основанные на вариантах использования.
- Инструменты отслеживания:
  - перехода требований в проект;
  - проекта в программный код.
- Для обеспечения тестирования.
- Для обеспечения технической поддержки.

При управлении большими проектами просто невозможно обойтись без некоторых компонентов CASE. Например, в обширном проекте совершенно необходим инструмент для управления конфигурациями.

2.5.3. Разрабатывать новые или покупать готовые решения?

Количество инструментов и приложений, предлагаемых на рынке, которые обещают помощь в разработке, все время возрастает. Например, планируя разработку приложения для проведения аукционов через Интернет, мы можем купить готовый пакет для разработки таких приложений или разработать свой собственный. Обычно принятие таких решений откладывается до момента, когда требования будут определены, но это решение является частью процесса управления проектом, поэтому оно рассматривается здесь.

Наиболее рациональным способом подготовки таких решений является составление списков затрат и сравнение величины затрат для различных вариантов. Это иллюстрируется в табл. 2.4, где рассматривается предполагаемая закупка программного обеспечения компьютерной графики Ajax для нашей видеоигры.

Таблица 2.4. Принятие решения о разработке или покупке инструмента (или приложения)

	Стоимость создания (в тысячах долларов)	Стоимость покупки	Комментарии
Инструмент	0	40	Покупка ПО корпорации Ajax
Средство 1	5	0	Настройка не нужна
Средство 2	10	1	Требуется настройка инструмента
Средство 3	15	15	Инструмент не помогает
	75	98	
	Не имеет смысла покупать инструмент (или приложение)		

Итоги по двум вариантам — с закупкой и без закупки — приведены в последней строке табл. 2.4. Необходимые средства машинной графики разбиты на три группы, и для каждой приведена отдельная оценка. А именно, пакет Ajax полностью поддерживает средство № 1 — отображение сцены как ее видит игрок. С другой стороны, Ajax не полностью поддерживает отображение трехмерных сцен (средство № 2), так что здесь придется делать доработки. Наконец, нам нужно моделировать освещение из определенного источника (средство № 3). Ajax не поможет, и данное средство придется запрограммировать. Таблица, подобная табл. 2.4, может быть приложением к плану проекта или к SDD. Более реалистичная таблица получится, если сравнить затраты в долговременной перспективе, то есть учесть затраты на поддержку и сопровождение. Чем больше нам придется делать самим, тем менее привлекательной выглядит закупка.

Многие решения, которые трудно принять, можно облегчить, если руководствоваться оценками затрат. Для команды и для других лиц, заинтересованных в проекте, равно как и для улучшения процесса, важно протоколировать такие решения (в частности, выбор между разработкой и закупкой).

2.5.4. Выбор языка программирования

Язык (или языки) программирования реализации должны быть выбраны вскоре после начала проекта. Иногда это решение прямолинейно, например, если организация использует только один язык или имеется возможность реализовать требования только на данном языке. Однако довольно часто существует возможность выбора из нескольких вариантов.

Примеры факторов и их весов, подсчетом которых можно выявить наиболее подходящий язык программирования, демонстрирует табл. 2.5. Так, язык № 1 набрал следующее количество баллов:

$$3 \times 8 + 8 \times 3 + 5 \times 2 + 1 \times 7 = 65.$$

Таблицы принятия решений не являются каким-то волшебством. Они просто расщепляют сложные задачи (например: «Какой язык выбрать?») на более простые (например: «Что лучше подходит для веб-приложений — Java или C++?»).

Такие расщепления предоставляют значительную стабильность. Однако решение, предоставляемое в результате анализа, сильно зависит от выбора весов, подбора факторов и сделанных выводов. Принятие таких решений лучше всего рассматривать вкупе с чьим-либо независимым взглядом.

Таблица 2.5. Пример метода выбора языка

Фактор	Вес (1–10)	Преимущество языка № 1 (от 1=меньше до 10=больше)	Преимущество языка № 2 (от 1=меньше до 10=больше)
Адаптирован к Интернету	3	8	2
Хорошо знаком команде разработчиков	8	3	9
Скорость компиляции	5	2	8
Скорость обработки данных на процессоре р	1	7	3
Итог		3×8+8×3+5×2+1×7 = 65	3×2+8×9+5×8+1×3 = 121

2.5.5. Документация

На стадии планирования команда должна решить, какой конкретно набор документации необходимо выработать для проекта. Альтернативы были рассмотрены в главе 1.

2.5.6. Службы поддержки

Проекту необходима поддержка со стороны системных администраторов, сетевых администраторов, администраторов баз данных, секретарей и т. п. Руководитель проекта должен обеспечить доступность этих служащих. В предложенном Хэмфри процессе TSP для этих целей выделяется специальный член команды, именуемый *руководителем поддержки*.

2.6. Подготовка плана-графика: планирование верхнего уровня

Используя всю накопленную к этому моменту информацию и уже проделанную работу, можно начать разрабатывать расписание проекта. Приведенную ниже форму расписания с откладыванием времени по горизонтали называют *диаграммой Ганта* (Gannt). На данном этапе диаграмма Ганта отражает только то, что известно о проекте на текущий момент (рис. 2.8). Заметим, что альтернативный вариант с тремя итерациями приводится в разделе 2.11.

ОДИН ИЗ СПОСОБОВ СОЗДАНИЯ ИСХОДНОГО ПЛАНА-ГРАФИКА

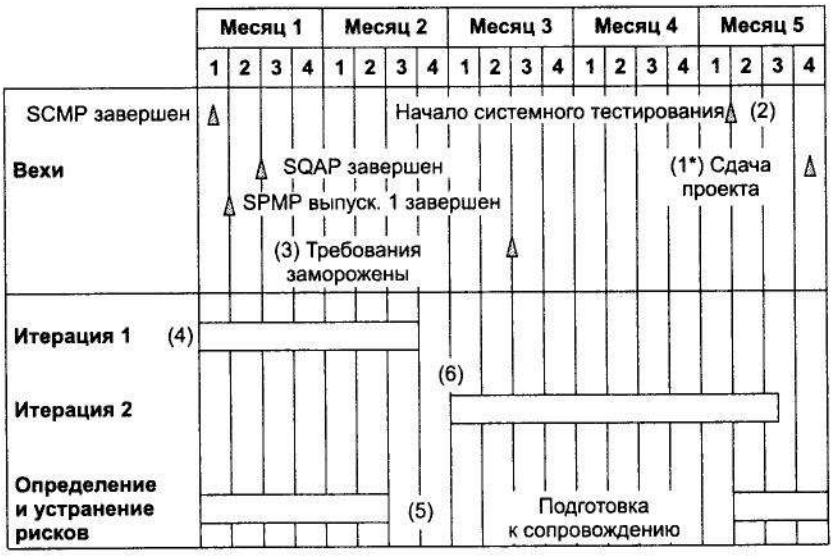
1. Определить основные вехи проекта (обычно включая дату сдачи проекта).
2. Пересмотреть их в связи с необходимыми для вас вехами:
   - например, системное тестирование начать задолго до даты сдачи проекта;
   - остальные шаги зависят от типа используемого процесса. Мы рассмотрим их для итеративного процесса.
3. Представить первую итерацию с использованием минимальной функциональности:
   - обычно ее оставляют очень простой, даже тривиальной в плане функциональных возможностей;
   - польза: налаживание процесса разработки.
4. Обозначить задачу по определению и устранению рисков.
   - Начиная с запуска проекта.
5. Представить дополнительное незагруженное время (например, неделю) около середины проекта.
6. Завершить составление плана-графика.

Даже в том случае, если итерационные методы позволяют постепенное пополнение требований, будет разумным назначить некоторую дату, после которой в проект не могут вноситься новые требования (шаг 3). Первая итерация должна быть достаточно скромной (шаг 4). Не следует пытаться объять необъятное. Даже самая тривиальная итерация имеет свое преимущество, заключающееся в тренировке участия команды в процессе, который может продолжаться на удивление долго. Помните, что обычно проще добавить новые возможности в скромный набор требований, чем исключить некоторые возможности из насыщенного набора требований. Дополнительным преимуществом использования достаточно небольших начальных итераций для учебных команд является тот факт, что командам еще только предстоит учиться документированию проектов и им не надо разрабатывать жесткие требования и документацию по проектированию.

Рекомендуется включить в расписание дополнительные периоды времени (шаг 6). Это делается из-за того, что мы не можем учесть действие всех внешних факторов (хотя и должны к этому стремиться). К тому же люди практически всегда растягивают решение задачи на все отведенное время. Если при этом в расписании не предусмотрено буферного времени, то все временные рамки наползают друг на друга, начинают перекрываться и образуют настоящую лавину. Одним из вариантов предупреждения возможных неожиданностей является *переоценивание* задач, то есть отведение под них заведомо большего количества времени. Другой вариант (см. рис. 2.8) заключается в создании буферных периодов в проекте (например, продолжительностью в одну неделю), во время которых не планируется решение каких-либо специальных задач. Намного проще изменить расписание внесением дополнительной буферной недели, чем переупорядочивать задачи.

Расписание становится все более детальным по мере продвижения проекта и проведения ревизий. В том числе, как только утверждается архитектура, мы определяемся с частными задачами. В это же время мы детально рассчитываем трудозатраты (кто, когда и над чем будет работать). В последующих главах будет

рассмотрено влияние каждого из шагов процесса разработки на расписание проекта. В некоторых случаях удается провести значительное распределение труда на самом высоком уровне. Например, если известно, что четыре человека полностью заняты с самого начала проекта, то всех их по возможности необходимо нанять. В этом случае результаты распределения работ в проекте сходны с приведенными на рис. 2.9.



**Номера показывают порядок, в котором создавалась эта таблица*

Рис. 2.8. Расписание с фиксированной датой сдачи: Порядок завершения (окончания)

Рис. 2.9. Уровень трудозатрат при фиксированной занятости

В расписании оставлены пробелы для непредвиденных обстоятельств в конце первой итерации на четвертой неделе второго месяца и после второй итерации на третьей и четвертой неделях пятого месяца. Никаких определенных мероприятий на эти недели не планируется. На данном этапе можно учесть отпуска служащих и т. д., из-за чего, собственно, количество занятых в проекте людей и не является постоянным.

Детали

Эту часть можно пропустить при первом чтении и вернуться к ней после прочтения последующих глав.

2.7. Интеграция унаследованных приложений

Основная часть работы программистов состоит не в разработке новых продуктов, а в расширении существующих систем или использовании их в новых приложениях. Такие уже существующие программы называют *унаследованными* приложениями. Иногда приложение появляется в результате исследовательского проекта, а не спланированной разработки, отвечающей профессиональным законам рынка. Источником для создания приложения может послужить прототип, неожиданно развившийся в прибыльный продукт, или приложение, созданное некоторое время назад без привлечения современных методов и стандартов документации. В подобных случаях возможна техническая недостаточность языка, документации, архитектуры и (или) кода. Однако такие приложения могут пользоваться спросом и обычно отлаживаются широким практическим использованием. Зачастую прекращение разработки таких приложений довольно неразумно, к тому же их замена непозволительно дорога.

Для того чтобы увеличить возможности унаследованной системы, мы можем либо добавить в нее новые возможности, либо отдельно создать требуемое приложение, которое будет использовать унаследованную систему. Эти альтернативы приведены на рис. 2.10. Распространенная проблема при использовании унаследованной системы — это трудности в понимании, что она делает и как она это делает. В основном эта проблема возникает при отсутствии подробной документации. Использование унаследованной системы становится намного проще, если при ее создании учитывалась практика разработки программного обеспечения.

Ругейбер и Уайт [94] приводят пример, в котором унаследованная телефонная система RT-1000 для автоматизированного распределения соединений вызывала некоторые проблемы, но приносила большой доход владельцу. Первоначально эта система стоимостью несколько миллионов долларов была создана командой из 70 разработчиков в результате более чем пяти лет работы. На ее за-

мену потребовалась бы гораздо большая сумма и трудозатраты. Компания решила восстановить и улучшить старую систему. Среди проблем RT-1000 были:

- отсутствие формального процесса при построении;
- отсутствие контроля версий;
- общая недостаточность документации;
- отсутствие полного списка тестов и их результатов, отсутствие автоматизации процесса тестирования;
- проблема 2000 года;
- реализация на нескольких языках, некоторые из которых устарели;
- некоторые побочные компоненты (например, система управления базой данных) больше не поддерживаются их производителями;
- заказчики использовали формы вывода RT-1000 для обеспечения ввода других программ (в частности, дисплеев), не ставя об этом в известность владельца RT-1000.

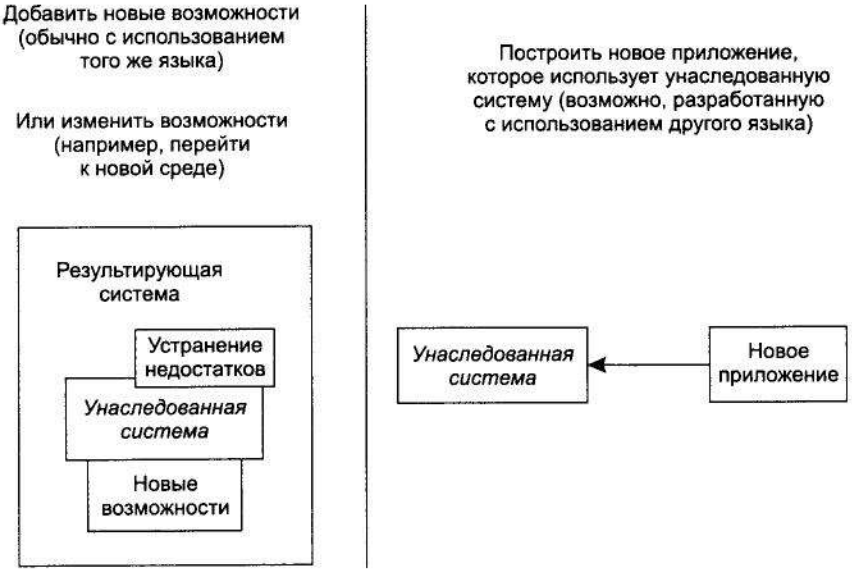


Рис. 2.10. Интеграция унаследованной системы наследования

Команда реставрационной разработки, работая в течение трех лет, проделала следующие операции:

- сократила количество открытых дефектов с 300 до 15;
- передала исходный код под управление конфигурациями;
- автоматизировала свыше 80 % существующих тестов;
- заменила или улучшила побочные компоненты или возобновила соглашения о технической поддержке;

- получила сертификат ISO-9001 для своего процесса;
- организовала постоянные посещения заказчиком организации-разработчика, таким образом улучшив отношения с заказчиком;
- добавила существенно новую функциональность;
- привела весь программный код к языку C.

Ругейбер и Уайт поведали о своих попытках использовать автоматический перевод Фортрана в C. Технически они добились успеха, однако код, полученный таким способом, было невозможно взять на сопровождение.

Сегодняшние приложения — это будущие унаследованные системы. Выясним, как этот факт влияет на создание новых приложений, таких как видеоигра *Встреча* в нашем примере. Мы сделаем эту игру легкой для изменения и расширения, поскольку концепции видеоигр часто меняются. Последовательно используя объектно-ориентированный подход, мы сделаем составные части игры ясными, не предполагая заранее, какие именно части будут меняться.

Предположим, что группа психологов узнает про игру *Встреча* и захочет использовать ее для исследования психологии играющих. В этом случае игра *Встреча* станет унаследованным приложением, а наш подход будет соответствовать изображенному на рис. 2.10 справа. Для этого мы снабдим игру *Встреча* подходящим программным интерфейсом (API). Такой интерфейс может состоять из следующих функций:

```
void runForDuration(int numMinutes) // выполнять игру заданное время
void runScenario(Scenario aScenario) // выполнить конкретный сценарий
flat getScore() // получить число очков-жизней игрока
void setKeystokeCounter(boolean aToggle) // считать нажатия клавиш
```

Чтобы решить задачу для психологов, нам нужно будет написать приложение-оболочку (или *адаптер*), которое будет использовать унаследованную систему с помощью указанного интерфейса. Образец проектирования Adapter рассматривается в главе 6.

Унаследованные приложения также рассматриваются в главе 10 данной книги в контексте задач сопровождения и поддержки.

2.8. Оценка стоимости: предварительные расчеты

2.8.1. Введение

Вкладчики в проект испытывают постоянный усиленный интерес к стоимости проекта. При неправильном подсчете себестоимости даже самый фантастичный продукт может потерпеть фиаско. Простейшим случаем оценки стоимости является ситуация, когда стоимость проекта изначально фиксирована и ни при каких обстоятельствах не меняется. Несмотря на то, что некоторые высокопрофессиональ-

ные организации преуспели в варьировании различных параметров (возможности, расписания, качества) для достижения заданной стоимости, стоимость проекта совершенно не обязана быть фиксированной. Представьте, например, что проект по созданию обещающего успех продукта выходит за рамки бюджета при 90-процентной готовности продукта. Вместо того чтобы бросить исходный проект, организация сделает все возможное для отыскания дополнительных фондов на завершение оставшихся 10 % работы. Даже в том случае, если стоимость проекта фиксирована, необходимо оценивать стоимость реализации заданного набора требований, чтобы быть уверенными в том, что она соответствует заявленной стоимости. Если соответствия нет, то необходимо изменить заявленную стоимость и заново провести оценку.

Процесс оценки стоимости (для фиксированных возможностей, уровня качества и расписания) часто начинается в начале проекта и продолжается даже на стадии написания программного кода. При открытии проекта у команды может быть весьма расплывчатое представление об его стоимости. Если оценку стоимости можно отложить до тех пор, пока проект не наберет полного хода, то так и нужно сделать, но всегда есть необходимость хотя бы грубой оценки диапазона стоимости уже на фазе анализа требований. Чем больше мы знаем о требованиях, предъявляемых к продукту, и чем дальше продвинуто проектирование, тем точнее мы можем оценить стоимость проекта (рис. 2.11).

Рис. 2.11. Разброс ошибок при оценке текущей стоимости проекта

Четырехкратная ошибка в оценке (см. рис. 2.11), взята из результатов исследования, представленного Боэмом [11]. Например, там указано, что для приложения, фактическая стоимость разработки которого составляет $100 000, оценка после составления концепции может варьироваться от $25 000 до $400 000. Чтобы уменьшить погрешность оценок на ранних фазах проекта, применяются различные приемы, но действительно точную оценку мы получаем в конце фазы реализации (когда большая часть денег уже потрачена!). Поскольку точную раннюю оценку дать практически невозможно, следует использовать диапазон, указывая минимальную и максимальную стоимость.

Некоторые люди удивляются, как вообще можно рассуждать о стоимости проекта, не имея детальных требований и не проводя проектирования. Но так же как и в других областях, это делается по аналогии. Например, грубую оценку стоимости возведения жилого дома можно получить следующим образом. Пусть известно, что в данном районе стоимость жилья составляет в среднем $100 за квадратный фут. Тогда дом жилой площадью 1000 квадратных футов обойдется примерно в $100 000.

На ранних стадиях проекта очень полезно делать оценку несколькими разными способами независимо, а потом сравнить результаты. Можно взять взвешенное среднее различных оценок, расставив веса в соответствии со степенью вашего доверия каждому методу.

Швейная машина или токарный станок — это сложные инструменты, которые бесполезны, если нет умеющего на них работать. Аналогично, совершенно невероятно, чтобы человек, делая первый раз раннюю оценку стоимости проекта, сразу получил верный результат. Но по мере накопления опыта и калибровки методов результаты улучшаются и точность оценок возрастает.

Типичная схема метода ранней оценки стоимости и сроков выполнения проекта имеет следующий вид.

1А. Используйте сравнение с предыдущими работами, чтобы непосредственно оценить стоимость и длительность проекта или количество строк кода.

И (ИЛИ)

1Б. Используйте метод функционального размера для оценки количества строк кода.

1Б.1. Вычислите приближенный функциональный размер.

1Б.2. Примените процесс уточнения.

2. Используйте оценку количества строк кода для расчета трудозатрат и длительности проекта с помощью формул COCOMO.

В следующем разделе приводится пример использования данных предыдущих проектов для оценки. Методология оценивания функционального размера (functional point) и конструктивной модели стоимости (COCOMO — Constructive Cost Model) описаны далее.

2.8.2. Оценка количества строк кода без учета функционального размера

В этом разделе описываются способы оценки количества строк кода на очень ранней фазе, задолго до проектирования и кодирования. По ходу проектирования можно применять более точные методы, основанные на результатах проектирования (см. рис. 2.11).

Несколько методов оценки, в особенности COCOMO, базируются на числе строк кода (LoC — Lines of Code). *Конструктивная модель стоимости* (COCOMO — Constructive cost model) была предложена Боэмом [11]. Может показаться, что на ранних стадиях проекта оценивать количество строк кода не имеет смысла, поскольку до кодирования еще далеко. Но если есть возможность сравнить один продукт с другим, то использование данной метрики становится вполне оправданным. Например, допустим, что у нас был проект по управлению спутниками и мы хотим сравнить наш новый проект, который предусматривает дополнительные возможности — мониторинг ураганов. Старый проект содержит три миллиона строк кода на Фортране. Дополнительная возможность потребует 100 000 строк на Фортране, что видно из сравнения с другими программами отслеживания ураганов. Если новый проект потребуется сделать на другом языке, то можно использовать средние по отрасли коэффициенты для пересчета количества строк кода.

Организации, которые имеют уровень больше 1 по CMM, обязаны учитывать фактические трудозатраты в человеко-часах на проделанную работу. Если таких данных нет, как в случае нашей видеоигры *Встреча*, придется сравнивать ее с другими играми. Получить фактические данные о трудозатратах в других компаниях трудно, если вообще возможно. Иногда данные частично публикуются, например в рекламном объявлении компании BugEye указано, что разработка ее новой игры велась два года. В объявлении может быть даже указано количество программистов, однако этим данным не следует слепо верить. Компании рассматривают свои знания о разработке как корпоративное достояние и часто скрывают фактические данные.

При отсутствии исторических данных придется сравнивать с аналогами. Например, для видеоигры аналогом может быть программа моделирования. Допустим, что у нас нет опыта программирования компьютерных игр, но есть небольшой опыт программирования моделирующих программ и знание языка Java. Тогда оценка количества строк кода может выглядеть примерно следующим образом:

«Однажды я написал неграфическую программу моделирования одиночного процесса на C++. Эта программа занимала от 4 до 8 страниц кода. Считая от 30 до 50 некомментированных строк на странице, имеем от 120 до 400 строк. Положим, что программирование на Java потребует такого же числа строк. В первом выпуске игры *Встреча* потребуется запрограммировать от 4 до 15 таких простых процессов, а также запрограммировать от 30 до 90 других компонентов сопоставимого размера, для того чтобы игра была интересной. Таким образом, получаем следующий диапазон: от 120 строк × 43 компонента до 400 строк × 105 компонентов, что округленно составит от 5 000 до 42 000 строк кода. Использование

графики учтем с помощью введения коэффициента, который может иметь величину от 1,5 до 4 в зависимости от сложности графических возможностей. Окончательно получаем оценку: от 7,5 тысяч строк до 170 тысяч строк».

ПРИМЕЧАНИЕ

Учебный пример в этой книге рассматривает прототип игры, который гораздо менее амбициозен по сравнению с проектом в этом примере оценки.

Эти данные нужно поместить в электронную таблицу, и мы сможем уточнять нашу оценку по мере продвижения проекта вперед. Заметьте, что наш диапазон 7,5–170 хорошо согласуется с рис. 2.11: на стадии формирования концепции следует ожидать, что минимум от максимума будет отличаться в 16 раз. Предшествующее вычисление является примером оценки по методу снизу вверх, поскольку мы получили оценку целого исходя из оценки составных частей.

Далее мы разберем пример оценки по методу сверху вниз, используя данные по отрасли (или, что предпочтительнее, исторические данные организации). Допустим, мы знаем, что в среднем по отрасли производство очень хорошей видеоигры требует усилий от 5 до 20 высококлассных программистов в течение 1–2 лет. Поскольку в нашем распоряжении имеется только одна десятая часть этих ресурсов, мы предположим, что и возможности нашей игры составят одну десятую от лучших промышленных образцов. Полагая производительность программистов от 5 до 25 строк (полностью отлаженных и документированных!) в день, получаем:

(1/10 возможностей) × (5–25 строк в день) × (5–20 программистов) × (1–2 года) × × (48–50 недель в году) × (35–60 часов неделю) ≈ 4,2–300 тысяч строк кода.

Полученный диапазон отличается от вычисленного ранее, но наша уверенность в правильности оценки порядка возрастает, поскольку мы использовали совершенно другой метод.

В Интернете имеются свободно распространяемые инструменты оценки, например http://www.construx.com.

Напомним, что PSP (см. главу 1) предполагает интенсивный сбор данных об индивидуально затраченных усилиях. Эта существенная практика вооружает как отдельных разработчиков, так и организации в целом историческими данными, необходимыми для последующих оценок.

2.8.3. Функциональный размер и количество строк кода

В 1979 году Альбрехт [2] предложил фундаментальное понятие *функционального размера* (FP — functional points), для измерения размера любого проекта независимо от проектирования. Метод измерения функционального размера состоит в единообразном измерении всех возможностей приложения и выражении размера приложения в виде одного числа. Это число можно далее использовать для оценки числа строк кода, стоимости и сроков проекта. Функциональный размер —

это весьма привлекательное понятие, поскольку оно претендует на то, чтобы измерить саму суть возможностей будущей программы. Однако нужно иметь определенный навык, чтобы применять этот метод аккуратно и последовательно.

Метод определения функционального размера состоит из следующих шагов.

Вычисление функционального размера: шаг 1

Идентифицируйте функции (например, «поиск данных», «отображение данных»), которые должно иметь приложение. Международная группа пользователей функционального измерения (IFPUG — International Function Point Users Group) [61] опубликовала критерии, по которым выделяются «функции» в этом смысле. Рассматривается функциональность на уровне пользователя, а не на уровне программного кода. Обычная функция соответствует обработке одной экранной формы.

Для нашей видеоигры, например, можно выделить следующие функции.

1. Установка характеристик главного персонажа игрока.
2. Встреча с внешним персонажем.

Вычисление функционального размера: шаг 2

Для каждой выделенной функции, исходя из тех данных, что указаны на рис. 2.12, сосчитайте количество факторов каждого типа. Ниже приведены объяснения для каждого фактора. Следует очень аккуратно следовать данным указаниям, в противном случае трудно получить надежную оценку.

- *Внешние входы.* Только такие входы, которые по-разному влияют на выполняемую функцию, считаются отдельными. Например, если функция заключается в сложении двух чисел, то она имеет один вход, а не два, то есть EI = 1. С другой стороны, если на вход можно подать букву A для выполнения сложения или букву S для выполнения вычитания, то такая функция будет иметь два входа, EI = 2.
- *Внешние выходы.* Отдельно считаются только выходы для существенно различных алгоритмов и нетривиальной функциональности. Например, вывод сообщения различными шрифтами нужно считать за 1. Сообщения об ошибках не учитываются. Диаграмма, представляющая данные, считается за 2 (1 для данных и 1 для формата диаграммы). Если выходные данные посылаются на существенно разные устройства (например, принтер и монитор), то они считаются отдельными выходами.
- *Внешние запросы.* Каждый независимый внешний запрос считается за 1.
- *Внутренние логические файлы.* Каждая уникальная логическая группа пользовательских данных, которая создается или поддерживается функцией, считается за 1. Комбинации таких групп не считаются, каждая группа данных должна учитываться один раз.
- *Внешние логические файлы.* Каждая уникальная логическая группа пользовательских данных, размещенная во внешних по отношению к приложению файлах, считается за 1.

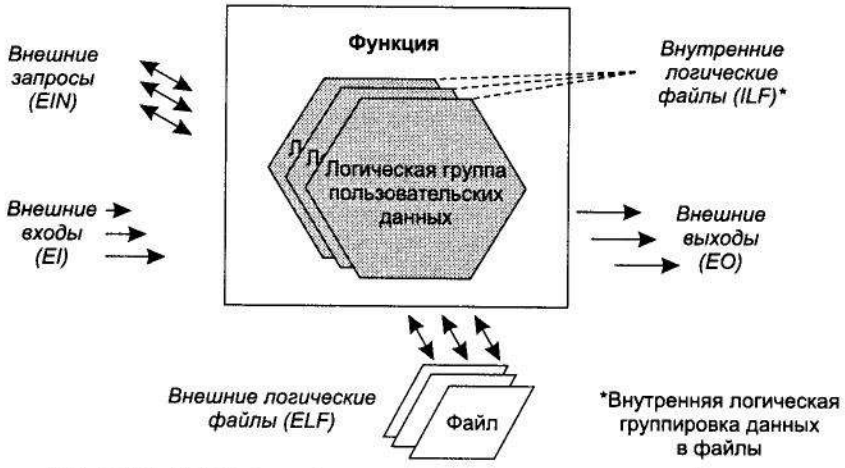


Рис. 2.12. Вычисление функционального размера для каждой функции

Вычисление функционального размера: шаг 3

Каждый из факторов, определенных на предыдущем шаге, умножается на коэффициент, определяемый сложностью данного фактора в приложении (табл. 2.6). IFPUG опубликовала детальное описание того, что следует считать «простым» и «сложным» в данной ситуации.

Таблица 2.6. Вычисление функционального размера до уточнения

Параметр	Просто					Сложно	
Внешние входы (EI)	×	3	или	4	или	6	=
Внешние выходы (EO)	×	4	или	5	или	7	=
Внешние запросы (EIN)	×	3	или	4	или	6	=
Внутренние логические файлы (ILF)	×	7	или	10	или	15	=
Внешние логические файлы (ELF)	×	5	или	7	или	10	=
							Итого:

Применяя этот процесс к двум выделенным функциям видеоигры *Встреча*, получим расчетные таблицы, приведенные в табл. 2.7 и табл. 2.8.

Таблица 2.7. Вычисление функционального размера до уточнения для функции «Установить характеристики персонажа игрока» игры Встреча

	Просто		Средне		Сложно		Промежуточные итоги	Всего
	число	фактор	число	фактор	число	фактор		
Внешние входы	1	3	1	4	1	6	13	
Примечание:	Имя		При готовности начать движение		Характеристики			

	Просто		Средне		Сложно		Промежуточные итоги	Всего
	число	фактор	число	фактор	число	фактор		
Внешние выходы	0	4	0	5	0	7	0	
Внешние запросы	0	3	0	4	0	6	0	25
Внутренние логические файлы	1	7	0	10	0	15	7	
Примечание:	Данные о персонаже игрока							
Внешние файлы интерфейса	1	5	0	7	0	10	5	
Примечание:	Данные о персонаже игрока							

Таблица 2.8. Вычисление функционального размера до уточнения для функции «Встреча с внешним персонажем» игры Встреча

	Просто		Средне		Сложно		Промежуточные итоги	Всего
	число	фактор	число	фактор	число	фактор		
Внешние входы	0	3	0	4	0	6	0	
Внешние выходы	1	4	0	5	0	7	4	
Примечание:	Результаты							
Внешние запросы	0	3	0	4	0	6	0	16
Внутренние логические файлы	1	7	0	10	0	15	7	
Примечание:	Данные о персонаже игрока							
Внешние файлы интерфейса	1	5	0	7	0	10	5	
Примечание:	Данные о персонаже игрока							

Полученные числа еще предварительные и нуждаются в уточнении, но они уже дают некоторые параметры для оценки работы. Полный неуточненный размер двух функций игры *Встреча* составляет 25 + 16 = 41.

Вычисление функционального размера: шаг 4

Теперь нужно определить веса для 14 *общих характеристик* проекта. Каждой общей характеристике присваивается вес от 0 до 5. Эти характеристики и выбранные веса применительно к проекту *Встреча* приведены в табл. 2.9 и 2.10. Веса указаны в форме диапазонов, что отражает нашу текущую неуверенность относительно функций приложения. Повторим еще раз, что назначение весов требует определенного опыта в использовании метода функционального размера. Например,

для характеристики № 6 требуется определить, насколько вероятна потребность в оперативном вводе данных. Применительно к функции установки характеристик персонажа игрока мы уверены, что оперативный ввод данных будет нужен, поэтому ставим наивысшее значение — 5.

Сумма выбранных значений общих характеристик лежит в диапазоне от 24 до 41.

Таблица 2.9. Общие характеристики проекта: факторы 1–7

	Иногда		*Средне*		*Всегда*
0	1	2	3	4	5
Отсутствует		*Редко*		*Часто*	

Фактор	В примере
1. Требуется резервное копирование (восстановление) данных?	0–2
2. Требуется обмен данными?	0–1
3. Используются распределенные вычисления?	0
4. Производительность важна?	3–4
5. Выполняется на сильно загруженном оборудовании?	0–1
6. Требуется оперативный ввод данных?	5
7. Используется много форм для ввода данных?	4–5

Таблица 2.10. Общие характеристики проекта: факторы 8–14

	Иногда		*Средне*		*Всегда*
0	1	2	3	4	5
Отсутствует		*Редко*		*Часто*	

Фактор	В примере
8. Поля базы данных обновляются оперативно?	3–4
9. Ввод, вывод, запросы сложные?	1–2
10. Внутренние вычисления сложные?	1–3
11. Код предназначен для повторного использования?	2–4
12. Требуется преобразование данных и установка программы?	0–2
13. Требуется множество установок в разных организациях?	1–3
14. Требуется поддерживать возможность настройки и простоту использования?	4–5

Вычисление функционального размера: шаг 5

Наконец, уточненный функциональный размер вычисляется по следующей формуле.

Уточненный функциональный размер = [Приближенный функциональный размер] × [0,65+0,01 × (Сумма общих характеристик)].

Суть этой формулы состоит в том, что если к приложению не предъявляется никаких специальных требований (все общие характеристики равны нулю), то неуточненный функциональный размер нужно уменьшить на 35 %. В противном случае неуточненный размер нужно увеличить на 1 % на каждую единицу значения общих характеристик.

Для нашего примера достаточно разумные значения общих характеристик приведены в табл. 2.9 и 2.10. Таким образом, окончательно получаем:

41 × [0,65 + 0,01 × (от 24 до 41)] = 41 × [от 0,89 до 1,06] ≈ от 36 до 43.

2.8.4. Преобразование функционального размера в количество строк кода

Функциональный размер может быть очень полезен, если он вычислен аккуратно. Например, функциональный размер может быть использован как относительная метрика для сравнения с предыдущими проектами. С помощью стандартных таблиц по функциональному размеру можно вычислить количество строк кода. В свою очередь, количество строк кода позволяет определить общую трудоемкость в человеко-месяцах и сроки проекта. Например, в [103] приведена следующая величина: 53 строки исходного кода на Java на одну единицу функционального размера. Используя этот коэффициент для игры *Встреча*, имеем (от 36 до 44) × 53 ≈ 1,9–2,3 тысяч строк на Java. Как и следовало ожидать, это значительно меньше предыдущих оценок, поскольку мы вычислили функциональный размер только для двух функций игры *Встреча*. В Интернете имеются свободно распространяемые инструменты для вычисления функционального размера [62].

2.8.5. Пример

Рассмотрим в качестве примера простую систему проката видеофильмов. Пусть мы имеем приложение, ориентированное на пользователя, которое позволяет узнать, имеется ли фильм в наличии, и взять его напрокат. Мы полагаем, что это приложение использует всего два файла: один для фильмов, другой для клиентов. Вычисление неуточненного функционального размера показано в табл. 2.11. Уточняющие коэффициенты приведены в табл. 2.12, и в сумме они дают 35. Формула функционального размера дает следующий результат:

Функциональный размер = [Неуточненный функциональный размер] × [0,65 + + 0,01 × (сумма общих характеристик)] = 33 × [0,65 + 0,01 × 35] = 33.

Таким образом, получается 33 × 53 = 1749 строк некомментированного исходного кода на Java.

Таблица 2.11. Вычисление функционального размера до уточнения для примера с прокатом видеофильмов

	Просто		Средне		Сложно		Промежуточные итоги	Всего
	число	фактор	число	фактор	число	фактор		
Внешние входы	2	3	1	4	0	6	10	
Пояснение:	Имя, телефон		Данные о видеофильме					

продолжение

Таблица 2.11 (*продолжение*)

	Просто		Средне		Сложно		Промежуточные итоги	Всего
	число	фактор	число	фактор	число	фактор		
Внешние выходы	0	4	1	5	0	7	5	
Пояснение:			Сумма					
Внешние запросы	0	3	1	4	0	6	4	33
Пояснение:			Наличие					
Внутренние логические файлы	2	7	0	10	0	15	14	
Пояснение:	Клиенты; видеофильмы							
Внешние файлы интерфейса	0	5	0	7	0	10	0	

Таблица 2.12. Уточняющие факторы для примера с прокатом видеофильмов

0	1	2	3	4	5
Отсутствует	*Иногда*	*Редко*	*Средне*	*Часто*	*Всегда*

Фактор	В примере
1. Требуется резервное копирование (восстановление) данных?	4
2. Требуется обмен данными?	0
3. Используются распределенные вычисления?	0
4. Производительность важна?	3
5. Выполняется на сильно загруженном оборудовании?	1
6. Требуется оперативный ввод данных?	5
7. Используется много форм для ввода данных?	3
8. Поля базы данных обновляются оперативно?	5
9. Ввод, вывод, запросы сложные?	2
10. Внутренние вычисления сложные?	1
11. Код предназначен для повторного использования?	3
12. Требуется преобразование данных и установка программы?	3
13. Требуется множество установок в разных организациях?	3
14. Требуется поддерживать возможность настройки и простоту использования?	2

2.8.6. Библиография

Метод вычисления функционального размера изложен Джонсом [68], имеющим практический опыт применения метода. См. также [28].

2.9. Оценка трудозатрат и длительности проекта по количеству строк кода

Как только мы оценили количество строк программного кода (либо с помощью единиц функционального размера, либо методом, приведенным в разделе 2.8.2), мы можем использовать его при оценивании затрат труда и продолжительности проекта. Боэм обнаружил [11], что на самом деле трудозатраты на разработку приложений растут быстрее, чем размер приложений. Для представления данного соотношения используется экспоненциальная функция со значением показателя, близким к 1,12. Модель Боэма также гласит, что длительность проекта возрастает экспоненциально вместе с прилагаемыми к проекту усилиями. Однако в данном соотношении значение экспоненты меньше единицы и составляет около 0,35. Это отражает тот факт, что после достижения некоторого значения («колена» на кривой 2 на рис. 2.13) дополнительные усилия только растягивают время, необходимое для завершения проекта. Это проиллюстрировано на рис. 2.13 (LOC обозначает количество строк программного кода).

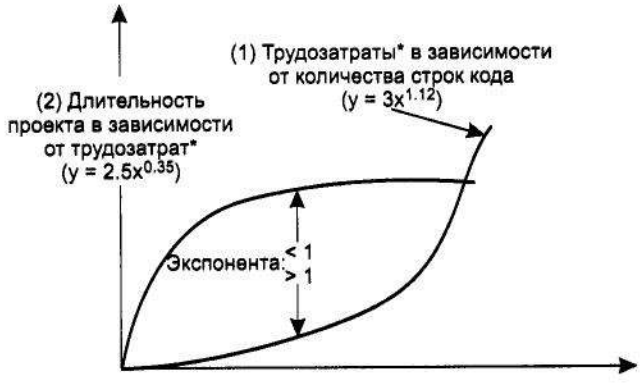


Рис. 2.13. Графики формул COCOMO

Используя данные из многочисленных проектов, Боэм оценил параметры рассматриваемых выше соотношений, предложив экспоненциальную зависимость. Полученные им формулы приводятся в табл. 2.13 (KLOC обозначает тысячу строк кода). *Органические* приложения — это самостоятельные приложения, такие как классические (например, без привлечения Web) текстовые редакторы или игра *Встреча* из нашего примера. *Встроенные* приложения являют собой интеграцию аппаратного и программного обеспечения (антиблокировочная система тормозов в автомобиле). *Промежуточные* — нечто среднее. Игра *Встреча* для случая игры в Интернете будет промежуточной: она уже не органическая, но еще и не так жестко встроена, как система управления тормозами.

Таблица 2.13. Основные формулы COCOMO (в соответствии с [11])

Программный проект	a	b	c	d
Органический	2,4	1,05	2,5	0,38
Промежуточный	3,0	1,12	2,5	0,35
Встроенный	3,6	1,20	2,5	0,32
Трудозатраты в человеко-месяцах = a × $KLOC^b$				
Длительность = c × $Трудозатраты^d$				

В модели Боэма утверждается, прежде всего, что для разных типов приложений длительность и трудозатраты по-разному зависят от размера приложений (отличаются множителем и показателем экспоненты). Например, разработка приложения в 20 000 строк потребует $2,4 \times 20^{1,05} \approx 51$ человеко-месяц, если приложение органично, и потребует $3,6 \times 20^{1,2} \approx 76$ человеко-месяцев, если это встроенное приложение.

Формула для длительности может быть прямо выражена через количество строк кода:

$$\text{Длительность} = c \times \text{Трудозатраты}^d = c \times (a \times \text{KLOC}^b)^d = c \times a^d \times \text{KLOC}^{bd}$$

На первый взгляд, формула Боэма для расчета длительности проекта может показаться несколько странной, так как зависимость между продолжительностью и прилагаемыми усилиями представляется намного проще. Если мы знаем, что работа требует 120 человеко-месяцев, и имеем в распоряжении 10 человек, не будет ли она сделана через 12 месяцев? Это может произойти, только если мы сумеем с пользой полностью занять 10 человек в проекте с первого и до последнего из 365 дней. Обычно это просто невозможно. Например, представьте себе первый день проекта: вы еще ничего не знаете о проекте, тогда какой же полезной деятельностью могут заниматься в этот день все 10 разработчиков? Таким образом, становится ясно, что при найме 10 разработчиков с первого дня проекта работа, рассчитанная на 120 человеко-месяцев, будет идти намного дольше 12 месяцев.

Формула Боэма обладает следующим интересным свойством: длительность проекта не зависит от количества привлекаемого в проект персонала! Она зависит только от объема работ. Вообще-то, формула предполагает, что в проекте имеется приблизительно необходимое количество персонала в каждый конкретный момент (один человек на первый день, 30 человек на сотый день, в соответствии с потребностями).

Модель Боэма интенсивно проверялась, получила широкое признание и уточнялась со временем. Тем не менее для ее эффективного использования нужен изрядный опыт и постоянная коррекция в соответствии со здравым смыслом.

Используя формулы Боэма [11] для нашей игры *Встреча* (рассматриваем две базисные функции), исходя из объема 4–300 тысяч строк программного кода, мы получим трудозатраты от 10 до 1000 человеко-месяцев и продолжительность от 6 до 36 месяцев (табл. 2.14).

ОДИН ИЗ СПОСОБОВ ПРЕДВАРИТЕЛЬНОЙ ОЦЕНКИ СТОИМОСТИ И ДЛИТЕЛЬНОСТИ ПРОЕКТА

1. Используйте метод функционального размера для оценки количества строк программного кода.
2. Используйте формулы Боэма для оценки требуемых трудозатрат.
3. Используйте оценку трудозатрат и формулы Боэма для оценки длительности проекта.

Таблица 2.14. Расчет модели примера Встреча с помощью COCOMO

	a	K	b	Аппроксимация
Трудозатраты				aK^b
Мал.	2,4	4,2	1,05	10
Бол.	2,4	300	1,05	1000
	c	**P**	**d**	**Аппроксимация**
Длительность				cP^d
Мал.	2,5	10	0,38	6
Бол.	2,5	1000	0,38	35

2.10. Командный процесс разработки программного обеспечения (TSP)

Никто не хочет постоянно иметь проблемы, обычные для всех проектов, связанных с программной разработкой. Особенно если эти проблемы можно решить заранее. Командный процесс разработки программного обеспечения (TSP) предлагает большое количество методов управления программным проектом, применимых к любой командной деятельности. TSP обеспечивает необходимую направленность и последовательность действий после анализа требований и по ходу фаз разработки проекта. Участники TSP должны быть подготовлены с точки зрения PSP. Метод основан на итерациях водопадной последовательности и требует, чтобы команда «запускала» каждую итерацию на совещании, где уже было бы известно количество предварительно определенных задач. Хэмфри предусматривает наличие множества различных сценариев. Так как фаза может проводиться в несколько итераций, то для нее, возможно, потребуется несколько «запусков». Перед запуском необходимо провести некоторые предварительные мероприятия:

- выбрать тип процесса, который будет использоваться;
- установить уровень качества;
- определить метод отслеживания уровня качества;
- определить, как команда будет принимать решения;
- определить, что делать, если не устраивает уровень качества:
  - идти на компромисс;
- определить, что делать, если не утвержден план:
  - идти на компромисс;

- ✦ определить роли в команде;
- ✦ распределить роли.

Хэмфри рекомендует, чтобы перед запуском каждой из фаз было определено следующее:

1) фиксированные цели команды;
2) определенные командные роли;
3) план развития процесса;
4) план качества;
5) план обеспечения проекта:
   компьютеры, ПО, персонал и т. д;
6) общий план разработки и план-график;
7) детальные планы для каждого разработчика;
8) оценка рисков проекта;
9) доклад о статусе проекта.

Многое из этого обсуждается и в примерах, и в документации IEEE, приводимых в этой книге.

TSPi — это разновидность TSP, модифицированного для проведения в рамках академического семестра. В TSPi распределяются следующие роли: *лидер команды*, *менеджер разработки*, *менеджер планирования*, *менеджер по качеству* (*процессу*) и *менеджер технической поддержки*. В этой главе мы употребляем термин «лидер» вместо «менеджер» для аналогичных ролей. *Менеджер технической поддержки* отвечает за предоставление и сопровождение всех инструментов и сред разработки, таких, например, как компиляторы.

Хэмфри предложил для TSPi свое расписание продолжительностью в один семестр. Это расписание содержит три итерации (Хэмфри назвал их «циклами») (рис. 2.14).

Идея заключается в том, что данные, полученные из каждого цикла, могут быть использованы при оценивании метрик для следующего цикла. Цикл 1 сравнительно продолжителен, так как на нем команда впервые проходит последовательность из семи фаз, показанную на рис. 2.14. Эта последовательность претендует на получение *минимального работающего подмножества функциональности конечного продукта*. Под цикл 3 отведено достаточное количество времени для завершения всей работы. Он начинается после относительно короткого среднего цикла. *Стратегия* (стоящая на первом месте на рис. 2.14) касается общей направленности, в которой команда собирается проводить цикл. Для этого потребуется обсуждение требований, концепции проектирования и общего плана сборки компонентов на самом высоком уровне. Затем все это преобразуется в жесткий план (2), фиксированные требования (3) и т. д. Хэмфри [55] предоставляет многочисленные детальные сценарии для сопровождения TSPi.

Рис. 2.14. Структура цикла TSPi

2.11. План управления программным проектом (SPMP)

План проекта должен быть составлен так, чтобы каждый знал, что и когда ему необходимо делать.

Существует множество стандартов для таких планов. Мы будем использовать стандарт IEEE 1058.1-1987 (утвержденный в 1993 году). Приведем оглавление Плана управления программным проектом (SPMP – Software Management Plan), определенное в IEEE 1058.1-1987 (мы используем его в конкретном примере в конце этой главы):

1. **Введение**
   - 1.1. Обзор проекта
   - 1.2. Результирующие артефакты проекта
   - 1.3. Развитие SPMP
   - 1.4. Ссылочные материалы
   - 1.5. Определения и аббревиатуры
2. **Организация проекта**
   - 2.1. Модель процесса
   - 2.2. Организационная структура
   - 2.3. Организационные рамки и взаимосвязи
   - 2.4. Ответственность за проект

3. **Управляющий процесс**
   - 3.1. Цели и приоритеты
   - 3.2. Допущения, зависимости и ограничения
   - 3.3. Управление рисками
   - 3.4. Механизмы мониторинга и контроля
   - 3.5. План расстановки кадров
4. **Технический процесс**
   - 4.1. Методы, инструменты и технологии
   - 4.2. Документация программного обеспечения
   - 4.3. Функции сопровождения проекта
5. **Распределение работ, план-график и бюджет**
   - 5.1. Распределение работ
   - 5.2. Зависимости
   - 5.3. Потребности в ресурсах
   - 5.4. Выделение бюджета и ресурсов
   - 5.5. План-график

Пункт 1.1 — «Обзор проекта» — должен определять проект, но не пытаться охватить все требования к нему (то есть описание его поведения). Сами требования будут приведены в Спецификации требований к программному обеспечению (SRS), которая будет рассмотрена далее в главах 3 и 4. Повторение этого материала в SPMP не обязательно и может нарушить целостность документации. Пункт 1.2 содержит список всех документов, исходных файлов и конечных программных продуктов, которые должны быть произведены. В пункте 1.3 описаны направления ожидаемого расширения и изменения SPMP. К этому времени уже должен быть разработан SCMP (см. главу 1), так что выпуск новых версий SPMP будет достаточно контролируемым.

Пункт 2.1 ссылается на тип процесса разработки, который будет использован (например, водопадный, спиральный, инкрементальный). Возможные организационные структуры мы уже рассматривали в разделе 2.3.2. В пункте 2.3 («Организационные рамки и взаимосвязи») представлены пути возможного взаимодействия между организациями. Все это зависит от заинтересованных в проекте сторон. Здесь может определяться, каким образом будет осуществляться взаимодействие между отделом разработки и отделом маркетинга, будут ли это регулярные встречи или переписка по электронной почте и т. д. Пункт 2.4 определяет границы ответственности, то есть кто за что отвечает. Например, за что несет ответственность координатор повышения эффективности команды при горизонтальной организации (см. рис. 2.5)? Отвечает ли он за общий успех проекта, предоставляет ли персональные рекомендации или занимается только руководством?

«Цели и приоритеты» (пункт 3.1) — раздел, в котором провозглашается рабочая философия проекта. Не все проекты имеют одинаковый приоритет. Для

нашей видеоигры главным приоритетом, вероятно, должно быть *создание среды, действительно захватывающей игрока*. Ведь если в результате никто не купит нашу игру, то все другие рассуждения бессмысленны. С другой стороны, для медицинских приложений первостепенным приоритетом является *надежность*. В других приложениях *возможность многоцелевого использования* может быть главенствующим приоритетом.

Управление рисками (пункт 3.3) было рассмотрено в разделе 2.4. Раздел «Механизмы мониторинга и контроля» (пункт 3.4) определяет, кто будет управлять, контролировать и (или) осуществлять проверку проекта, а также предписывает, как и когда это должно быть сделано. Например, вышестоящий управляющий должен быть в курсе развития проекта, и для этого мы должны в этом разделе определить, каким образом его информировать. «План расстановки кадров» содержит информацию о том, кто и какое место занимает в проекте. В частности, в пункте 2 может быть задано присутствие в проекте *Менеджера* с конкретными обязанностями, тогда как в пункте 3.5 будет сказано, что это место займет Альберт Смит.

«Технический процесс» в пункте 4 накладывает ограничения на языки и используемые инструменты («в этом проекте должны быть использованы язык Java от корпорации Sun версии 1.2.1 и Rational Rose версии 1»). Также пункт 4 может содержать информацию о повторно используемых требованиях и использовании таких техник, как образцы проектирования (главы 5 и 6). «Функции сопровождения проекта» — раздел, посвященный действиям для поддержания процесса разработки, таким как управление конфигурациями и контроль качества. Если же функция поддержки представлена в различных документах (например, в плане управления конфигурациями или в плане качества), то в этом пункте будут ссылки на эти документы. В противном случае мы полностью специфицируем функции поддержки.

Пункт 5.1 («Распределение работ») описывает то, как работа должна распределяться и предоставляться после выполнения. Поскольку архитектура приложения еще не утверждена, то первая версия этого пункта, конечно, будет несколько поверхностной. Например, на этой стадии проекта мы можем заявить, что технический контролер ответственен за создание SDD. Однако мы не можем углубиться в подробности, из-за того что проект еще не определен. Рабочий пакет все больше обрастает деталями в последующих версиях SPMP.

В пункте 5.3 («Потребности в ресурсах») оцениваются трудозатраты, аппаратное и программное обеспечение, необходимые для сборки и технической поддержки приложения. Также здесь могут быть приведены результаты оценки стоимости. Этот пункт пересматривается по мере прогрессирования проекта и становится более точным и детализированным.

Пункт 5.4 («Бюджет и выделение ресурсов») распределяет ресурсы между различными частями проекта в течение всего его жизненного цикла. Он состоит в основном из оценки стоимости человеко-дней, однако включает в себя и стоимость техники и программного обеспечения.

Наконец, пункт 5.5 завершает SPMP созданием расписания, определяющим, как и когда должны быть выполнены различные этапы процесса. Это обсуждалось нами в разделе 2.6.

2.12. Управление проектом и качество

Каждый хочет работать только над «хорошими» проектами. Однако мы не можем себе этого позволить, пока не определимся с тем, что мы будем считать «хорошим» проектом. Для этого нам необходимо определить метрики, с их помощью оценить наши проекты, а затем улучшать их, пока они не станут «хорошими».

2.12.1. Метрики процесса

2.12.1.1. Введение

Напомним, что 5-й уровень CMM (см. раздел 1.8.3) требует постоянного самостоятельного улучшения процесса. Хотя некоторые организации и работают на этом уровне, им необходимо помнить свои основные цели. Для того чтобы улучшить процесс управления проектом, мы должны будем измерить его эффективность с помощью метрик процесса. Эти метрики позволят нам измерить эффективность организации нашего процесса, включая последовательность шагов, предпринимаемых в ходе проекта. Мы также по отдельности измерим эффективность анализа требований, проектирования, программирования и тестирования.

2.12.1.2. Примеры

Одной из простейших метрик является *степень обнаружения дефектов* для заданной фазы обнаружения и заданной фазы появления. Например, «степень обнаружения дефектов, равная 0,2 дефекта на 100 требований на стадии реализации» означает, что на стадии реализации 500 требований в одном из них был обнаружен дефект.

Когда степени обнаружения дефектов сравниваются с нормами организации, происходит оценка всего процесса в целом, а не только конкретного проекта. Пример проекта с накопленными данными о дефектах на каждой фазе представлен в табл. 2.15. Для простоты мы опустили фазы тестирования и технической поддержки, которые могли бы дополнить общую картину.

Таблица 2.15. Процент дефектов на каждой фазе

	Фазы, на которых были обнаружены дефекты (В этом проекте/Норма)		
Фаза, содержащая дефекты	**Детальные требования**	**Проектирование**	**Реализация**
Детальные требования	2/5		
Проектирование	0,5/1,5	3/1	
Реализация	0,1/0,3	1/3	2/2

Обратим внимание на фазу детальных требований в табл. 2.15. В результате инспектирования на данной фазе были обнаружены два дефекта на 100 требований, что несколько меньше нормы для данной организации, составляющей пять дефектов на 100 требований. Просматривая столбец «Детальные требования», можно заметить, что степень обнаружения дефектов в требованиях в нашем процессе меньше нормы на всех фазах проекта. Это говорит о том, что наш проект, а возможно, и используемый нами процесс, сравнительно эффективны в отношении задания качественных требований.

Результат в отношении дефектов проектирования заключается в следующем. Мы обнаружили больше дефектов проектирования с помощью инспектирования непосредственно на той фазе, когда они были произведены, и на более поздних фазах было обнаружено меньше дефектов. Поскольку чем позже обнаруживается и исправляется дефект, тем дороже это обходится, данный факт может означать, что наш проект и, может быть, наш процесс лучше, чем в целом по организации.

Чтобы завершить таблицу, следовало бы добавить аналогичные данные по дефектам, собранным на фазе тестирования и в определенный период времени (скажем, три месяца) после поставки.

Перечислим подходящие метрики процесса, включая метрики, связанные с дефектами и обсужденные выше. Сравните нижеследующие величины с нормами компании для подобного процесса.

1. Количество дефектов на тысячу строк программного кода, выявленных в течение 12 недель после сдачи проекта.
2. Отклонения в расписании на каждой фазе:

   (*Фактическая длительность – Плановая длительность*) / *Плановая длительность*.
3. Отклонения в стоимости:

   (*Фактическая стоимость – Плановая стоимость*) / *Плановая стоимость*.
4. *Общее время проектирования / Общее время программирования*:

   Должно быть не менее 50 % (Хэмфри).
5. Степени появления и обнаружения дефектов на некоторой фазе:

   Например, «Один дефект на класс на фазе детального проектирования».

Обратите внимание, что только сравнение с данными по организации или по отрасли позволяет измерить процесс: сами по себе полученные числа ни о чем не говорят. Например, если в нашем проекте оказался выявленным один дефект на тысячу строк кода в течение шести месяцев после поставки, в то время как норма по организации составляет 1,3 дефекта на тысячу строк кода, выявленных за тот же период, то наш процесс можно считать улучшенным. Однако такой вывод можно сделать, только если мы располагаем данными по нескольким проектам, проведенным по одному процессу, и знаем средние величины.

Дополнительную информацию по метрикам процесса можно найти в [59]. В следующем разделе обсуждается использование измеренных показателей для улучшения процесса.

2.12.2. IEEE 739-1989 SQAP: часть 2

Как уже упоминалось в главе 1 (раздел 1.6.5), все соображения, связанные с качеством в проекте, могут быть полностью отражены в таком документе, как SQAP. Представим вторую половину SQAP. Иногда удобнее, если части SQAP просто ссылаются на другие документы:

7. **Тестирование**

   Может ссылаться на STD
8. **Отчеты о проблемах и коррекционная деятельность**
9. **Инструменты, технологии и методики**

   Может ссылаться на SPMP
10. **Контроль программного кода**

    Может ссылаться на SCMP
11. **Контроль носителей**
12. **Контроль поставщиков**
13. **Сбор, сопровождение и хранение протоколов**
14. **Обучение**
15. **Управление рисками**

    Может ссылаться на SPMP

Попытка дублирования этих документов или их частей нарушила бы наше правило ведения целостной документации. В примере в конце главы обсуждаются оставшиеся незатронутыми до этого момента темы.

ОДИН ИЗ СПОСОБОВ СБОРА МЕТРИК ПРОЦЕССА

Последовательность действий, которые необходимо предпринимать на протяжении жизненного цикла проекта с целью его улучшения, может быть, например, такой.

1. Выявить и определить метрики, которые будут использоваться командой на каждой фазе, включая:
   - время, затраченное на исследование, реализацию и анализ результатов;
   - размер (например, количество строк кода);
   - количество дефектов, обнаруженных в модуле (например, количество строк кода), и источник обнаружения дефекта;
   - самостоятельная оценка качества друг у друга по шкале от 1 до 10.
2. Задокументировать это в SQAP.
3. Собирать статистику на каждой фазе.
4. Решить, где разместить метрические данные:
   - в ходе развития проекта;
   - SQAP? SPMP? Приложения?
5. Назначить разработчиков, ответственных за сбор данных на каждой фазе:

   ответственный за качество или лидеры фаз (например, ответственный за проектирование).
6. Запланировать обзоры полезных в дальнейшем метрических данных:
   - определить, когда и как организовать улучшение процесса обратной связи.

Примерный вариант накапливаемых о проекте данных применительно к детальным требованиям содержит табл. 2.16. Процесс накапливания детальных требований обсуждается также в главе 4. Таблица 2.16 в принципе применима и к другим фазам проекта. Приведенные в таблице числа взяты для примера и ни в коем случае не являются нормативными. Сравнение с нормативами организации выявило недостатки в процессе *проведения совещаний*. В таблице команда оценила проведение совещаний на 2 по десятибалльной шкале. Выяснилось (в таблице это не отражено), что процесс проведения совещаний можно улучшить, если бы предложения, высказываемые на них, были бы более завершенными.

Таблица 2.16. Сбор метрик процесса на каждой из фаз

Документация требований: 200 детальных требований	**Совещание**	**Исследование**	**Реализация**	**Персональная проверка**	**Инспектирование**
Затрачено часов	0,5×4	4	5	3	6
от общего времени	10 %	20 %	25 %	15 %	30 %
от общего времени: норма для организации	15 %	15 %	30 %	15 %	25 %
Самостоятельная оценка качества 1–10	2	8	5	4	6
Количество дефектов на 100				5	6
Количество дефектов на 100: норма для организации				3	4
Затрачено часов на детальное требование	0,01	0,02	0,025	0,015	0,03
Затрачено часов на детальное требование: норма для организации	0,02	0,02	0,04	0,01	0,03
Улучшение процесса	Улучшить заготовки, представляемые на совещании		На реализацию затрачено времени на 10 % больше		
Итого:					
Продуктивность: 200/22 = 9,9 детальных требований в час.					
Вероятная норма оставшихся дефектов: 6/4 × [при норме для организации 0,8 на 100] = 1,2 на 100					

Другая проблема возникла во время стадии выполнения требований, когда начинается непосредственная работа по фиксации требований. Уровень дефектности выше нормального (5 вместо 3), а самооценка по качеству несколько ниже средней (4). Проведенное сравнение с нормами компании показало, что необхо-

димо выделить больше времени на стадию выполнения требований, тем самым уменьшив количество дефектов и улучшив самооценку. Читатель из этого процесса может заключить, что стандарт для учета частей каждой фазы является фундаментом для наших попыток использовать метрики. В связи с этим мы и занимаемся учетом *детальных требований*, что подробнее будет изложено далее в главе 4.

Остаточная степень дефектности относится к количеству дефектных детальных требований на 100 оставшихся в документации требований после завершения фазы детальных требований. Это позволяет нам судить о том, насколько эффективной была наша деятельность. Остаточная степень дефектности получается в результате вычисления пропорции остаточной степени дефектности в организации за последние несколько проектов. Последнюю мы получаем расчетом процента дефектных детальных требований в предыдущих проектах после завершения создания документации по требованиям, то есть на стадиях проектирования, реализации и тестирования. Полученное отношение 6/4 — это *степень дефектности требований, полученная для данного проекта*, отнесенная к *средней степени дефектности требований*. Основная идея заключается в возможности сравнения степеней дефектности на предыдущем и последующем шагах. Более надежное предсказание можно получить, учитывая степень дефектности требований, полученную для предыдущих шагов (в нашем случае — в ходе персонального инспектирования). Для этого необходима линейная регрессия.

Даже не имея исторической базы, команда должна заранее определиться с тем, какими могут быть и какими должны быть значения метрик. Имея эти предварительные договоренности, команда будет работать лучше, к тому же больше будет вероятность запоминания результатов работы. Полученные данные станут основой для создания базы данных о проектах компании. Управление всем этим не является технически сложной задачей, однако этим приходится заниматься параллельно с выполнением других не менее важных задач. По этой причине организация четкой ответственности и регулярное проведение обзоров метрических данных прорабатываются на столь ранней стадии процесса. Как будет видно из следующего раздела, именно процесс улучшения процесса обратной связи определяет отличие великих компаний от просто хороших.

2.13. Улучшение процесса и модель зрелости возможностей

Как описывается в предыдущей главе, наивысший уровень CMM достигается постоянным самоулучшением процесса с использованием метапроцесса. Однако как добиться того, чтобы это работало на постоянной основе? Существует два уровня улучшения процесса. Первый улучшает способ разработки приложений компанией. Второй улучшает процесс, используемый в конкретном проекте.

2.13.1. Улучшение процесса, используемого в организации

Общее улучшение процесса в первую очередь требует классификации типов работ и процессов. Классификация типов работ всецело зависит от компании. Например, в специализированном магазине это могут быть: «проверка двигателей», «компьютерная диагностика», «статистика». Таким образом, это означает, что все приложения, имеющие отношение к этому магазину, могут быть отнесены к одному из трех перечисленных типов. Благодаря такой системе можно эффективно собирать различную статистику. Пример сбора данных приводится в табл. 2.17.

Таблица 2.17. Пример сравнивания процессов

Приложения управления двигателем	Процесс		
	Водопадный	Спиральный, 2–4 итерации	Спиральный, 5–10 итераций
Среднее по компании количество дефектов на тысячу исходных строк кода к моменту сдачи внесенных на фазе:			
требований	4,2	3,2	2,4
архитектуры	3,1	2,5	3,7
детального проектирования	1,1	1,1	2,2
реализации	1,0	2,1	3,5
Итого:	9,4	8,9	11,8

Такие же таблицы могут быть созданы для приложений «компьютерной диагностики» и «статистики». Остается один вопрос: «Как мы можем использовать это для улучшения процесса?». Один из методов — попытка использования лучших частей различных процессов. Например, в соответствии с табл. 2.17, меньше всего дефектов получается во время реализации для водопадного процесса, использованного для приложений, связанных с «проверкой двигателей». Однако спиральный процесс с тремя итерациями дал наилучший общий результат. Это подсказывает нам решение проблемы. Необходимо модифицировать наш спиральный процесс из Требования (Т)-Архитектура (А)-Детальное проектирование (Д)-Реализация (Р)-Т-А-Д-Р-Т-А-Д-Р в процесс Т-А-Д-Р-Т-А-Д-Т-А-Д-Р, который сохраняет спираль с тремя итерациями, но сокращает количество итераций реализации.

2.13.2. Улучшение процесса для текущего проекта

Команда может оценивать свою деятельность применительно к каждому процессу в проекте, а затем использовать эти оценки в качестве обратной связи в последующих процессах даже для текущего проекта. Пример оценки деятельности команды представлен в табл. 2.18.

ОДИН ИЗ СПОСОБОВ УЛУЧШИТЬ ПРОЦЕСС (ПРОЕКТ) С ПОМОЩЬЮ ОБРАТНОЙ СВЯЗИ

1. Осуществить декомпозицию процесса или субпроцесса на Подготовку, Выполнение и Анализ выполнения.
   - Добавить Исследование, если проводится изучение нового материала.
2. Отмечайте затрачиваемое время, оценивайте уровень качества для каждой составляющей по шкале от 1 до 10, подсчитывайте количество дефектов.
   - Старайтесь поддерживать кривую.
3. Вычисляйте отношение качества к затрачиваемому (в процентах) времени.
4. Если это возможно, сравнивайте продуктивность команды с имеющимися данными статистики.
5. Используйте полученные данные для улучшения следующего субпроцесса.

   Отмечайте меньшие значения первыми, например низкий показатель *Качество / (% затрачиваемого времени)*.

Таблица 2.18. Измерение продуктивности команды на каждой фазе

	Для каждой части		
	Подготовка	**Выполнение**	**Анализ выполнения**
Процент затраченного времени	45	30	25
Качество (от 0 до 10)	6	2	6
Если низкое, расследование причин	–	Расследовать	–
Качество/(% затраченного времени)	0,13	0,07	0,24
Если низкое, расследование причин	Расследовать	Расследовать	–
Типичное?	Нет (Джо потерял спецификации)	Да	Да
Меры	–	Отвести на фазу выполнения на 20 % больше времени, вычтя его из других фаз	–

Самый низкий уровень качества и качества, отнесенного ко времени в процентах, получен для фазы *выполнения*. Предлагаемое решение проблемы заключается в выделении на эту фазу в будущем большего объема времени за счет сокращения длительности других фаз. Команда должна оценить это предложение и решить, удовлетворительно ли оно, или имеются другие пути решения.

Компании не могут достичь пикового уровня производительности за одну ночь, поэтому для достижения верхних уровней предпринимаются последовательные попытки. Вместо того чтобы добиваться превосходного уровня во всем сразу, используются PSP, TSP и CMM, благодаря которым осуществляются промежуточные улучшения в деятельности и технологии. Уровни CMM обеспечивают ступенчатую структуру для улучшения процесса. Компании необходимо достичь уровня 2, прежде чем перейти на уровень 3, и т. д.

2.14. Вспомогательные средства и методы управления проектом

В этом разделе кратко обсуждается расстояние как фактор формирования команды разработчиков. Это сопровождается разговором об *экстремальном программировании* — примере попытки управления процессом разработки приложений. Последний раздел посвящен использованию отбраковки как основного метода принятия решений. Этот метод может оказаться полезным при принятии быстрых решений в сложных ситуациях.

2.14.1. Распределенные и международные команды

Вполне естественно, что менеджеры стараются извлечь пользу из таланта программистов, живущих по всему миру. Это не только снижает затраты на производство программного обеспечения, но и позволяет улучшить его качество. Почасовая оплата удаленных программистов незначительна по сравнению с проблемами связи, возникающими ввиду физической отдаленности. Интернет сделал отдаленность менее проблематичным фактором. С другой стороны, возрастает потребность в более продолжительном взаимодействии с заказчиком. Для многих приложений общение «лицом к лицу» становится просто необходимым. Достоинства и недостатки конфигураций команд с удаленными участниками заключаются в следующем.

- Расположение в одном офисе:
  + идеально для общения в группе;
  – продуктивность работы не оптимальна.
- Расположение в одном городе, но в разных офисах:
  достаточная степень общения.
- Расположение в одной стране, но в разных городах:
  – трудности с постоянным общением;
  + одинаковая культура.
- Расположение в разных странах:
  – общение затруднено;
  – культурные связи проблематичны;
  + продуктивность работы оптимальна.

2.14.2. Экстремальное программирование

Экстремальное программирование — это методика управления проектом и разработкой, предложенная Кентом Беком [7]. Этот раздел включен в книгу для ознакомления читателя с разнообразными методами и техниками, использовавшимися

в течение долгого времени. Кроме того, в нем излагаются дополнительные идеи по использованию экстремального программирования в различных обстоятельствах. Интересными особенностями экстремального программирования являются упор на непрерывную взаимосвязь как внутри организации разработчиков, так и с заказчиком, радикальная простота (использование наиболее простого решения) и парное программирование. При парном программировании разработчики работают в парах за компьютерами, тем самым исключается изоляция. Сравнение некоторых особенностей экстремального и неэкстремального программирования приводится в табл. 2.19 [5].

Таблица 2.19. Сравнение экстремального и неэкстремального программирования

Неэкстремальное	Экстремальное
Заказчик изолирован	Заказчик находится в команде
Предварительное проектирование	Проектирование на ходу
Продукт создается в том числе и с расчетом на будущее	Создание актуального продукта
Допускается сложность	Предельная простота
Задачи распределены	Задачи ставятся самостоятельно
Разработчики изолированы	Парное программирование
Нечастая интеграция	Непрерывная интеграция
Ограниченное общение	Непрерывное общение

Мы уже рассматривали некоторые их этих приемов в других контекстах. Например, Министерство обороны США давно имеет своих представителей в командах, выполняющих крупные заказы. Экстремальное программирование идет дальше, делая представителя заказчика одним из участников разработки (автор однажды был таким участником). В принципе, это отличная идея, хотя она влечет некоторые юридические проблемы, которые не всякая организация может легко решить. Возможно, самой характерной чертой метода Бека является программирование в парах, при котором разработчики работают только вдвоем за одним компьютером. Фактически это своеобразная форма непрерывного инспектирования. Андерсон и другие [5] сообщают превосходные результаты в своем исследовании применения экстремального программирования в корпорации Крайслер.

Постоянная интеграция и постепенное добавление программного кода весьма полезны в определенных обстоятельствах, особенно в поздних частях проекта, и напоминают технологию «синхронизация и стабилизация» корпорации Microsoft.

Любой разработчик понимает противоречие между предельной простотой с одной стороны и общностью для повторного использования с другой. В примере этой книги упор сделан на повторное использование. В многих ситуациях, однако, предельная простота может быть предпочтительнее. Например, вам, наверное, будет не по себе, если подрядчик будет тратить заметное время на наладку своих инструментов и опробование технологий для своих будущих работ, в то время как он строит ваш дом. Вы с этим согласитесь, только если эта деятельность заметно улучшит ваш дом.

2.14.3. Принятие решений с помощью отбраковки

Во время выполнения проекта часто возникает перегрузка. Например, список пожеланий «что нужно сделать» разбухает очень быстро и кажется, что может разрастаться безгранично. Обычный путь борьбы с этим явлением — устанавливать приоритеты. Однако на этом идеальном пути также очень быстро возникает перегрузка. Например, в списке «что нужно сделать» имеется 100 пунктов, а время есть только на выполнение 20 из них. Тогда упорядочивание по приоритету всех 100 пунктов — пустая трата времени. Для решения этой проблемы можно использовать отбраковку.

Суть метода отбраковки состоит в том, чтобы любой вопрос анализировать не более чем два раза. После этого мы занимаемся только пунктами из категории *Немедленно*. Если (если!) они все будут выполнены, то переходим к рассмотрению категории *Обычное*, и т. д. Если угодно, пункты можно расставить по приоритетам внутри категорий. Таким образом, мы не теряем время на определение точного порядка действий, которые никогда не будем выполнять. По данным «Business Week» [19], такой прием обработки сообщений о найденных ошибках использовался в Microsoft во время отладки Windows 2000. Принцип метода отбраковки:

```
IF принадлежит к числу наиболее важных
    поместить в категорию НЕМЕДЛЕННО
ELSE
    IF можно проигнорировать, не меняя существенно проект,
        поместить в категорию В ПОСЛЕДНЮЮ ОЧЕРЕДЬ
    ELSE
        поместить в категорию ОБЫЧНОЕ
```

2.15. Подведение итогов

Основной идеей данной главы является то, что способ управления проектом так же важен, как и технологические аспекты проекта. Йордон [112] пошел еще дальше, назвав управление проектом «серебряной пулей», которую профессиональные программные разработчики искали для решения проблем запаздывания, выхода за рамки бюджета и низкого качества программных продуктов. SPMP служит основным навигатором при управлении проектами. Ключевым аспектом является оценка расходов на проект. Этот процесс требует непрерывного отслеживания в течение всего жизненного цикла проекта. Перечислим основные моменты этой главы.

- Управление проектом — «серебряная пуля»?
- «Человеческие» аспекты не уступают техническим.
- Специфицируйте SPMP.

- Идентифицируйте и устраняйте риски.
- Оценивайте затраты, используя различные методы:
  - готовьтесь к исправлениям и доработкам;
  - используйте на этом этапе диапазоны значений.
- Составьте план-график проекта с соответствующей детализацией.
- Поддерживайте баланс между стоимостью, графиком работ, качеством и функциональностью.

Руководство по учебному проекту. План управления программным проектом (SPMP) для видеоигры Встреча

В этом разделе объясняется, как применяются на практике изложенные в данной главе принципы. В качестве примера используется учебный проект *Встреча*.

Предполагается, что до того, как приступить к составлению SPMP, команда провела уже по крайней мере одно совещание, на котором обсудила проект в целом, выбрала лидера команды (Эд Браун) и составила SCMP и SQAP.

Этап 1. Подготовка к совещанию по планированию проекта

Незадолго до совещания Эд просмотрел заголовки разделов SPMP, предложенные IEEE (см. оглавление IEEE 1058.1-1987 в разделе 2.11), и набросал черновик материала по каждому выбранному разделу. Для случая видеоигры *Встреча* лидер команды решил, что нужны разделы Цели и приоритеты (пункт 3.1 IEEE 1058.1-1987), Ответственность за проект (основные и вспомогательные роли, а также их ответственности, пункт 2.4), Управление рисками (пункт 3.3) и План-график (пункт 5.5). Кроме того, Эд написал небольшой абзац для раздела 1.1 (Обзор проекта). Он не заполнял план набора кадров (то есть не стал определять, кто будет играть какую роль), поскольку решил, что пусть лучше это сделает сама команда во время совещания на добровольной основе. Он запланировал заполнить оставшиеся разделы после совещания. По электронной почте Эд спросил, нет ли добровольца на проведение оценки стоимости проекта, поскольку это техническая работа, которую лучше делать одному, максимум вдвоем.

Заполняя раздел Цели и приоритеты, Эд перечислил возможные варианты, а не выбрал сразу самое важное, поскольку не хотел, чтобы группа считала, что все решено без ее участия и проект катится по железным рельсам. В качестве наиболее приоритетной цели он предложил такие варианты: «достичь поставленных целей по качеству», «сделать что-нибудь такое, чем мы сами будем пользо-

ваться» (это его личный выбор) и «уложиться в план-график». Он был совершенно уверен, что группа согласится на организацию по ролям (см. раздел 2.10), поэтому он включил это в проект решения.

Эд запросил по электронной почте мнение участников о рисках, которые угрожают проекту, и попросил прислать ему мнения за 48 часов до совещания в форме, приведенной в табл. 2.2. Карен выразила сомнение в способностях группы в программировании на Java, и описала этот риск так подробно, как смогла. Кроме того, она навела справки о компаниях, предоставляющих услуги по краткосрочному обучению. Пошаговый план устранения риска она включила в письмо, которое послала Эду. Халл Фурнес затронул вопрос о наложении изображений в Java и послал сообщение об этом риске Эду. Последний включил все присланное в заготовку SPMP и назначил приоритеты.

Затем Эд составил следующую повестку дня совещания.

Совещание состоится в помещении 397 с 10:00 до 11:30 в субботу 11 сентября.

1. *Назначение секретаря совещания (5 мин — 10:05).*
2. *Утверждение повестки дня (5 мин — 10:10).*
3. *Обзор разделов SPMP, представленных Эдом (25 мин — 10:35).*
4. *Назначение ответственных за другие разделы SPMP (20 мин — 10:55).*
5. *Определение процесса проведения обзоров — по почте или на совещании (5 мин — 11:00).*
6. *Мозговой штурм по выявлению дополнительных рисков (10 мин — 11:10).*
7. *Обзор намеченных действий (5 мин — 11:15).*
8. *Разное (10 мин — 11:25).*

Эд послал повестку и свою заготовку SPMP участникам команды за два дня до совещания и попросил ознакомиться с материалами до совещания. В его версии SPMP присутствовали все заголовки, предлагаемы IEEE.

Этап 2. Начальное совещание по планированию проекта

На совещании Эд попросил Ферна вести протокол принимаемых решений и намечаемых действий, а Элу предложил следить за регламентом. Было решено, что на будущих совещаниях эти две роли будут поочередно играть все участники. Большинство предложений Эда было принято. Было предложено несколько изменений в плане-графике. Халл настоял на включении резервной недели, на которую не намечается никаких работ. Карен заметила, что на неделю, предшествующую началу сессии, ничего назначать нельзя. Затем была дискуссия об использовании простого водопадного процесса с целью избежать переработки документов, но это было признано нереальным. Ферн настаивал на использовании инкрементального процесса, потому что хотел поскорее начать программировать, но это предложение не получило поддержки, поскольку еще не была утверждена архитектура.

Участники пришли к мнению, что *качество* — это та область, в которой им больше всего необходимо приобретение опыта, поэтому было решено, что *достижение заданных параметров качества* является их наиболее приоритетной целью. Участники осознали, что создать сверхпривлекательную игру в отведенное время нереально и что им нужно сосредоточиться только на минимальных возможностях. Когда команда подошла к распределению ролей, Карен первой вызвалась быть ответственной за проектирование. Нашлись три добровольца на роль ответственного за реализацию и ни одного на роль ответственного за качество. Эд предложил компромисс, чтобы разделить эту роль между двумя людьми и они бы поменялись ролями в середине семестра. Другие роли были заняты, и Эд напомнил обязанности каждой роли и определил, кто кого будет замещать в случае необходимости.

Дискуссия по поводу написания SPMP вышла за отведенные временные рамки, но была продуктивной, поэтому Эд не стал прерывать ее. Было решено, что еще двое участников, кроме Эда, будут писать SPMP и что все остальные будут инспектировать их работу, поскольку трудно организовать в короткое время коллективное написание документов. Через 10 минут Эд заметил, что команда начинает обсуждать мелочи и прекратил дискуссию, пообещав, что все мелкие вопросы будут решены заочно по электронной почте. На совещании было принято решение, что писатели должны закончить сочинение своих разделов SPMP к вечеру пятницы, Эд должен собрать все воедино и разослать в 15:00 в субботу. Все должны прислать свои замечания к 15:00 воскресенья, и Эд должен учесть их в окончательной версии документа. Эд должен будет решить, нужно ли проводить еще одно совещание по данному вопросу, и сообщить об этом всем до 20:00 в воскресенье. Совещание, если понадобится, будет проведено в помещении 283 в понедельник в 11 часов.

Ферн напомнил, какие решения приняты, главным образом кто что должен написать и к какому сроку. После этого совещание закончилось и все разошлись.

Этап 3. Завершение составления SPMP

Выписывая детали документа, команда осознала, что многие вопросы не были обсуждены на совещании, в том числе Механизмы мониторинга и контроля (пункт 3.4 в IEEE 1058.1-1987, см. раздел 2.11). Этот раздел писал Халл и в первом варианте он предусмотрел множество совещаний для обзора состояний проекта. Но другие участники команды посчитали, что не нужно проводить так много совещаний. Прочитав несколько предложений, Эд завершил дискуссию по электронной почте, предложив делать обзор проекта на еженедельных совещаниях, а также на каждом совещании, посвященном началу фазы (которые он также постарался приурочить к еженедельным совещаниям). Команда согласилась с этим предложением. Кроме того, на случай, если понадобятся дополнительные совещания, был зарезервирован еще один день недели, на который они будут назначаться.

Упражнения

Ответы и подсказки для упражнений, помеченных символами «о» или «п», приводятся в конце этой главы.

Вопросы для проверки

П2.1[п]. Назовите пять стадий процесса планирования программного проекта.

П2.2°. Как распределить обязанности между тремя очень опытными инженерами при выполнении работы, требующей три человеко-месяца? Опишите тремя предложениями, как будут приниматься решения.

П2.3°. Какую эффективную форму организации стоит использовать для выполнения работы в 4000 человеко-месяцев силами 100 разработчиков? Опишите четырьмя предложениями.

П2.4°. Перечислите по меньшей мере два последствия отсутствия письменного плана проекта.

П2.5°. Укажите те части SPMP, которые вы, возможно, не сможете составить на данной фазе. Это нужно сделать в форме ссылок на те разделы, к которым необходимо будет вернуться на более поздних стадиях работы.

П2.6°. Почему нужно составлять план выявления и устранения рисков при составлении плана проекта?

П2.7°. Опишите тип проекта, для которого составление плана выявления и устранения рисков может быть неоправданно.

П2.8°. Обычно оценка стоимости важна. Можете ли вы указать обстоятельства, при которых не стоит выполнять оценку стоимости проекта?

П2.9°. Укажите один положительный и один отрицательный аспект использования метода функционального размера при оценке стоимости.

П2.10°. Укажите один положительный и один отрицательный аспект использования метода Боэма при оценке стоимости.

П2.11°. Что такое метрики процесса? Приведите три примера. Для чего их используют?

Упражнения в команде

К2.1. (SPMP) Разработайте SPMP для своего проекта. Используйте (с изменениями и улучшениями) стандарт IEEE, как показано в примере ниже. Внесите по крайней мере две итерации в план-график. Грубо оцените размер проекта.

Прежде чем начинать работу, оцените количество дефектов на страницу, которое, как вы думаете, обнаружит команда во время окончательного обзора. Ведите учет времени, затраченного каждым участником и всей командой на выполнение следующих стадий: исследование, подготовка документов, обзор (включая инспектирование). Определите фактическую плотность дефектов (число дефектов на страницу). Оцените эффективность своей команды на каждой стадии числом

от 0 до 10. Подведите итог и определите, как может быть улучшен командный процесс. Можно использовать дополнительные метрики, если они повышают эффективность в текущем и последующих проектах.

Критерии оценки.

1. Степень ясности плана и его приложений («Отлично» — все совершенно ясно и конкретно).
2. Степень реалистичности плана и его приложений («Отлично» — все цели достижимы и нет противоречий).
3. Степень полноты плана и его приложений («Отлично» — план включает > 95 % необходимых конкретных деталей).
4. Степень неизбыточности плана и его приложений («Отлично» — план включает < 5 % ненужных деталей).
5. Полезность вашей самооценки.

К2.2. (SQAP) Создайте реалистичный план контроля качества для вашего проекта. Измерьте время, затраченное на этот план отдельными разработчиками и командой в целом.

Предоставьте соответствующие данные измерений и самооценки, как в К2.1.

Критерии: такие же, как и для К2.1.

Подсказки

П2.1. Посмотрите на схему процесса управления проектом в разделе 2.1.4.

Ответы

П2.2. Предполагая, что участники команды заинтересованы в проекте, разумно использовать модель команды равных, выделив одного человека в качестве лидера проекта. Он будет задавать тон, принимать волевые решения и следить за расписанием. Остальные берут на себя каждый определенную часть процесса.

П2.3. Проект такого размера требует заметных усилий на организацию. Должна быть установлена иерархия управления (по крайней мере руководитель проекта/руководители по направлениям/разработчики). Относительно небольшая команда лучших разработчиков должна быть назначена на подготовку и анализ требований и разработку архитектуры. Работа должна быть разбита на несколько частей, и руководители по направлениям должны вести каждую часть, на каждую часть можно назначить своих разработчиков. Руководители направлений в своих подразделениях могут использовать модель команды равных.

П2.4. Без письменного плана даже самые компетентные разработчики будут терять время и выполнять пустую работу. Независимо от того, написан план на бумаге или нет, он все равно существует, может быть в головах некоторых разработчиков. Если план не написан, он должен быть проговорен устно. Хотя это звучит сильно, на самом деле это предельно непрофессиональный подход,

и маловероятно, что в результате получится профессиональный продукт. Устная передача информации тем плоха, что теряются следующие вещи:

- какой процесс разработки используется;
- кто, что и когда должен делать;
- какие риски грозят в будущем. Без плана идентификации и устранения рисков проект сдается на милость неизвестных обстоятельств в будущем.

П2.5. Можно представить черновик плана-графика, но в нем невозможно предусмотреть детали. Дело в том, что детальный план можно составить только зная, как работа разделена на части, а это неизвестно в данный момент, потому что еще не проведено проектирование архитектуры.

П2.6. Любой проект имеет риски, связанные со стоимостью, расписанием и качеством. Ранняя идентификация рисков оставляет время на принятие мер по их предупреждению, что уменьшает ущерб от неблагоприятных событий.

П2.7. Может случиться так, что в процессе идентификации рисков удастся выявить только малое их число, причем таких, для которых меры по предупреждению были бы приняты в любом случае. В этом случае время окажется потрачено зря, и единственным полезным результатом будет успокоение.

П2.8. Может возникнуть искушение ответить на этот вопрос так: небольшой проект, который нужно выполнить любой ценой. Нужно согласиться, что время, затраченное на оценку, забирает слишком большую долю из времени, отпущенного на работу. Но даже это не оправдывает отказ от оценки стоимости. Можно считать, что время, необходимое для оценки объема работы, пропорционально объему работы, так что объем проекта не имеет большого значения. Отказ от оценки может быть оправдан, только если тот, кто оценивает, совершенно не умеет этого делать. Действительно, если три инженера две недели учатся оценивать стоимость проектов, а потом применяют полученные знания, чтобы оценить один двухмесячный проект, то игра не стоит свеч. В больших проектах эти две недели немного значат.

П2.9. Преимущество: возможность сделать очень раннюю оценку стоимости. Недостаток: потенциально высокий разброс в значении функционального размера, особенно если применяющий не имеет достаточно опыта.

П2.10. Преимущества: возможность оценить сразу сроки и трудозатраты; точность формул. Недостаток: метод основан на измерении количества строк кода, а эта величина часто неизвестна.

П2.11. Метрики процесса измеряют эффективность процесса. Например, мы можем сравнить эффективность водопадной и спиральной модели для данной организации в данное время. Следующие метрики позволяют это сделать.

- (Общее время проектирования)/(Общее время разработки).

  Обычно должно быть 0,5 или чуть больше.
- (Число дефектов на тысячу строк)/(Среднее по компании).
- (Доля недокументированных требований)/(Среднее по компании).
- [(Фактическая продолжительность проекта)/(Плановая продолжительность проекта)]/(Среднее по компании).

Последние три метрики являются метриками процесса, и могут быть отдельные проекты, которые имеют значительные отклонения от среднего.

Пример 1. План управления программным проектом (SPMP) для видеоигры Встреча

Утверждаю

_____Дата_____

05.01.98 Эд Браун: Создание первой версии

02.02.98 Халл Фурнас: Рецензирование и различные предложения по улучшению

16.05.98 Эд Браун: Детализирован план-график, добавлены ссылки на валидацию и верификацию

29.05.98 Эд Браун: Проверка для выпуска

1. Введение

1.1. Обзор проекта

[Примечание для студентов. Каждый проект имеет уникальную историю и концепцию. Данный раздел SPMP — самое подходящее для этого место. Объем раздела зависит от того, для кого предназначен SPMP. Если SPMP используется только в команде, то этот раздел должен быть очень кратким и отражать только консенсус, к которому пришла команда, определяя назначение проекта. Если круг читателей более широкий, то здесь нужно описать общий контекст проекта.]

Данный проект организован для разработки видеоигры, называемой *Встреча*. Игра будет разработана в несколько этапов, поскольку заказчик намерен специфицировать игру поэтапно с учетом результатов предыдущего этапа. Первые версии создаются в целях обучения, чтобы разработчики могли попрактиковаться в технологии разработки, и в качестве базы, на которой студенты могут создавать свои собственные игры. Последующие версии, как ожидается, будут либо свободно распространяемыми, либо коммерческими играми.

1.2. Результирующие артефакты проекта

Следующие материалы должны быть поставлены в указанные сроки.

Версия 1 (прототип) с документацией — вторая неделя второго месяца.

Версия 2 с документацией — третья неделя пятого месяца.

Документация включает в себя SPMP, SQAP, SVVP, SCMP, SRS, SDD, STD (с использованием стандартов IEEE), исходный код, компилированный байт-код, План сопровождения программного обеспечения и Руководство пользователя.

Аббревиатуры определены в разделе 1.5.

1.3. Развитие SPMP

[Примечание для студентов. В этом разделе объясняется, как будет поддерживаться и развиваться данный документ. Этот документ обязательно будет изме-

няться (например, появится более детальный план-график), поэтому необходимо определить ответственного за поддержание данного документа в актуальном состоянии.]

Данный документ поддерживается лидером проекта. Лидер проекта должен поместить данный документ под управление конфигурациями и обязан поддерживать документ в актуальном состоянии, еженедельно внося необходимые изменения. Данный SPMP в основном следует стандарту IEEE 1058.1-1987.

1.4. Ссылочные материалы

Все необходимые стандарты IEEE опубликованы в сборнике стандартов IEEE, редакция 1997 года.

Данный документ должен быть согласован с корпоративным документом «Генеральный план достижения уровня 5 CMM».

Основное руководство: Software Engineering: an Objected-Oriented Perspective. E. Braude, Wiley, 2000.

1.5. Аббревиатуры

CI — Configuration Item. Элемент конфигурации.

CMM — Capability Maturity Model. Модель зрелости возможностей.

IEEE — Institute of Electrical and Electronics Engineers. Институт инженеров по электротехнике и радиоэлектронике.

QA — Quality Assurance. Контроль качества.

SEI — Software Engineering Institute. Институт технологий разработки программного обеспечения.

SCMP — Software Configuration Management Plan. План управления конфигурациями программного обеспечения.

SPMP — Software Project Management Plan. План управления программным проектом (данный документ).

SRS — Software Requirements Specification. Спецификация требований к программному обеспечению.

SDD — Software Design Document. Проектная документация программного обеспечения.

STP — Software Test Plan. План тестирования программного обеспечения.

2. Организация проекта

2.1. Модель процесса

Первые две версии этого проекта будут выполнены с использованием спирального процесса разработки, по одной итерации на версию. Итерации проводятся в соответствии с USDP. Согласно USDP, итерации классифицируются на начальные итерации, итерации проектирования, конструирования и перехода. Первая итерация будет состоять только из анализа и планирования требований, вторая итерация будет первой в серии итераций проектирования. Это составит версию 1 игры *Встреча*. Количество последующих итераций и состав версии 2 будут определены после того, как заказчик ознакомится с версией 1.

2.2. Организационная структура

Организация проекта *Встреча* в рамках корпорации Gaming Industries Consolidated представлена на рис. 2.15. Проект организован как команда равных с назначением ролей. Роли следующие: лидер команды, ответственный за конфигурацию, ответственный за качество, ответственный за требования, ответственный за проектирование и ответственный за реализацию. Кроме того, имеется роль ответственного за связи с отделом маркетинга и с лабораторией технологии программирования. Все эти роли показаны на рис. 2.16. В проекте *Встреча* будет проводиться инспектирование всей работы, как это определено в SQAP. Каждый участник команды будет проводить инспектирование работы других участников (см. рис. 2.16). Инспектирование будет происходить либо групповое, либо, если не будет хватать времени, индивидуальное автором и тем, кто его замещает.

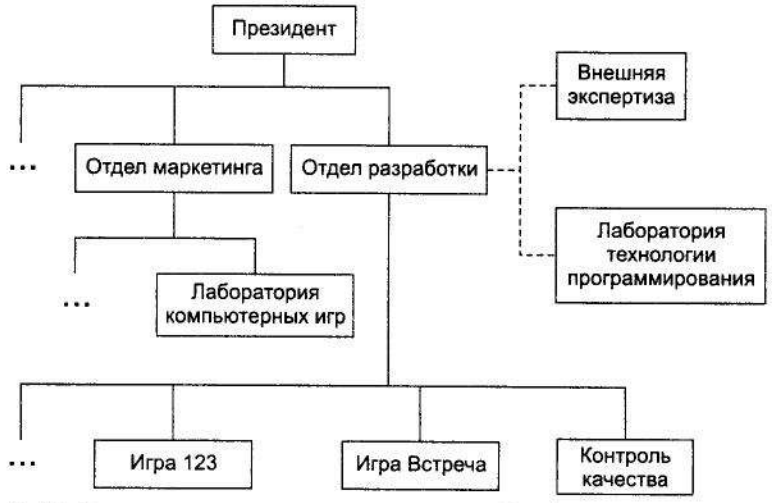


Рис. 2.15. Организационная структура корпорации Gaming Industries Consolidated

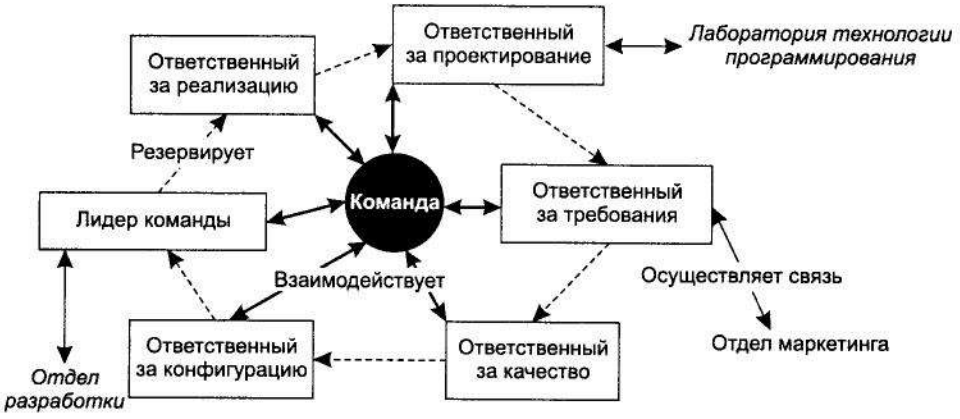


Рис. 2.16. Организация проекта Встреча

2.3. Организационные рамки и взаимосвязи

[Примечание для студентов. Укажите людей и организации, с которыми должна взаимодействовать команда.]

Команда проекта должна взаимодействовать со следующими людьми и организациями: отдел разработки, отдел маркетинга, лаборатория компьютерных игр, внешние эксперты и лаборатория технологии программирования.

2.4. Ответственность за проект

Ответственность участников проекта показана в табл. 2.20.

Ответственность за документ подразумевает следующее:

- документ должен быть создан вовремя;
- лидер команды определяет, кто пишет документ;
- документ поддерживается в актуальном состоянии.

Таблица 2.20. Ответственность участников проекта Встреча

Участник	Лидер команды	Ответственный за конфигурацию	Ответственный за качество	Ответственный за требования	Ответственный за проектирование	Ответственный за реализацию
Отвечает за связь	Отдел разработки			Отдел маркетинга	Лаборатория технологии программирования	
Отвечает за документ	SPMP	SCMP	SQAP, STP	SRS	SDD	Исходный код

3. Управляющий процесс

3.1. Цели и приоритеты

[Примечание для студентов. Здесь устанавливается относительный приоритет расписания, бюджета и возможностей приложения. Возможности подразделяются на степень выполнения требований, уровень качества и пригодность для повторного использования. Строго говоря, повторное использование должно предусматриваться требованиями.]

Высший приоритет имеет достижение заданного уровня качества. На втором месте по приоритетности стоит выполнение проекта в срок. Третий приоритет имеет выполнение максимального количества требований. Четвертый приоритет имеет создание классов, которые можно повторно использовать в других видеоиграх. Привлекательная видеоигра ожидается только начиная с версии 3.

3.2. Допущения, зависимости и ограничения

[Примечание для студентов. Здесь перечисляются предположения и допущения о внешних по отношению к проекту событиях, которые могут повлиять на проект.]

Нет.

3.3 Управление рисками

[Примечание для студентов. Определяйте риски как конкретные неприятности, которые могут произойти. Не ограничивайтесь общими словами. Например, утверждение «недостаточные навыки программирования на Java» само по себе не описывает суть проблемы. Иногда проект можно успешно выполнить и с недостаточными навыками.]

Форма для идентификации и обработки рисков показана в табл. 2.21. Каждое совещание по проекту должно включать в повестку дня пункт «мозговой штурм с целью выявления рисков». Риск № 1 «Наложение изображений» связан с возможностями манипулирования изображениями на языке Java. Предположим, что никто в команде не пробовал накладывать изображения. Накладывать изображения необходимо, поскольку изображения персонажей должны двигаться на фоне изображения зоны. Никто в команде не пробовал накладывать изображения так, чтобы не было видно прямоугольной границы накладываемого образа. Мы не знаем, легко ли это сделать, трудно или вообще невозможно.

Риск № 2, «недостаточные навыки программирования на Java», отражает тот факт, что 40 % команды не имеют достаточного опыта программирования на Java для того, чтобы реализовать движение и взаимодействие изображений персонажей. Мы предполагаем, что понадобится масштабировать игровое пространство, и в этом также нет опыта. Мы не знаем, сможем ли мы реализовать на Java те возможности, которые имеет в виду заказчик, а если сможем, то сколько это потребует времени. Это обстоятельство может серьезно повредить проекту.

3.4. Механизмы мониторинга и контроля

[Примечание для студентов. Как правило, имеет смысл запланировать регулярное проведение совещаний (обычно раз в неделю). Если окажется, что нечего обсуждать, такое совещание легко отменить. С другой стороны, назначить совещание внезапно довольно трудно, потому что участники команды могут быть заняты другими делами. Даже в тех командах, которые вместе работают ежедневно, регулярные совещания нужны, чтобы избежать «разброда и шатаний».]

Вся команда будет встречаться на совещании в начале каждой фазы (определение требований, проектирование, реализация) каждой итерации. Должны проводиться еженедельные совещания по проекту по вторникам с десяти утра до полудня. Следует принять все меры к тому, чтобы на этих совещаниях рассматривались сразу все общие для команды дела. Участники команды должны зарезервировать время по пятницам с 9 до 11 для проведения дополнительных совещаний, если понадобится. Лидер команды должен предупредить участников о проведении дополнительного совещания не позднее 16:30 в четверг.

[Примечание для студентов. В настоящем проекте нужно поручить разным людям готовить различные отчеты. Типичный список отчетов приведен в табл. 2.22.]

Таблица 2.21. Таблица рисков для игры Встреча

Номер риска	Название риска (детали оговорены отдельно)	Вероятность (L, 1–10)	Ущерб (I, 1–10)	Стоимость устранения (R, 1–10)	Приоритет (11 – L) × × (11 – I) × R	План устранения (уменьшения)	Ответственный разработчик	Дата завершения работ
1	Наложение изображений	3	10	1	(11 – 3) × × (11 – 10) × × 1 = 8	Экспериментирование с наложением изображений в Java	ПР	02.01.99
2	Недостаточные навыки программирования на Java	9	10	7	(11 – 9) × × (11 – 10) × × 7 = 14	Посещение всеми внесенными в список учебных курсов, начинающихся 05.01.99 в Ultra Training Corp. Получение сертификатов ABC — 01.03.99 и сертификатов XYZ — 15.04.99	ХЛ	04.15.99
3	...	...	...	...	...	...	...	...

Таблица 2.22. Программа мониторинга и контроля

Ответственность	Лидер команды	Связь с отделом маркетинга	Контроль качества	Связь с лабораторией компьютерных игр	Устранение рисков
Отчет на еженедельном совещании	×	×	×		×
Циркулярная рассылка отчета каждую неделю			3*		
Циркулярная рассылка отчета каждые две недели				4*	
Циркулярная рассылка отчета каждый месяц	1*	2*			
*Формы отчетности					
1	См. CI 34: «Форма ежемесячного отчета о состоянии проекта»				
2	См. CI 87: «Форма ежемесячного отчета о состоянии маркетинга»				
3	См. CI 344: «Форма еженедельного отчета по качеству»				
4	См. CI 48: «Форма двухнедельного отчета по результатам работы лаборатории компьютерных игр»				

3.5. План расстановки кадров

Назначение ролей указано в табл. 2.23. Каждый участник команды имеет дополнительные обязанности по резервированию и инспектированию (см. рис. 2.16).

Таблица 2.23. Расстановка участников проекта Встреча

Имя	Лидер команды	Ответственный за конфигурацию	Ответственный за качество	Ответственный за требования	Ответственный за проектирование	Ответственный за реализацию
Эд Браун	×					×
Эл Пруит		×				
Ферн Трайфил			×			
Халл Фурнес				×		
Карен Петерс					×	
Осуществляет связь с	Отдел разработки			Отдел маркетинга	Лаборатория технологии программирования	

4. Технический процесс

[Примечание для студентов. В этом разделе описывается технология, используемая в проекте, но только в тех аспектах, которые не являются спецификацией требований.]

SRS описывает некоторые аспекты требуемого технологического процесса. Здесь рассматриваются те аспекты, которые не установлены явно в SRS.

4.1. Методы, инструменты и технологии

В проекте *Встреча* для проектирования используется Rational Rose, а реализация ведется на языке Java. Во всех случаях используется объектно-ориентированный подход. Для документирования используется Javadoc настолько широко, насколько это возможно (подробнее см. SRS). Используемая модель процесса описана в разделе 2.1.

4.2. Документация программного обеспечения

См. SQAP, раздел 4.2.

4.3. Функции сопровождения проекта

На все время проведения проекта будет привлечен на полставки технический специалист по поддержке инструментов.

5. Распределение работ, план-график и бюджет

5.1. Распределение работ

Распределение работ показано на рис. 2.17. В нижней строке показано количество человеко-месяцев, приходящихся на данный месяц.

[Примечание для студентов. Пока не начато проектирование, еще нельзя конкретно указать, кто какую часть будет выполнять. Конкретные имена будут добавлены после того, как будет проведено проектирование для различных конфигураций.]

	Месяц 1				Месяц 2				Месяц 3				Месяц 4				Месяц 5			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Вехи	△ SCMP										Тестирование закончено △									
		△ SQAP								△ Требования заморожены							Поставка △			
	△ SPMP выпуск 1																			
Задачи	Итерация 1																			
	Риски: идентификация и устранение								Итерация 2											
Э. Браун	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Э. Пруит	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		
Ф. Трайфил	1	1	1	1	1	1	1	1	1	1	1	1	1							
Х. Фурнес	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
К. Петерс	1	1	1	1	1	1	1	1						1	1	1	1	1	1	1
Ф. Смит (технический специалист)	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5	.5
Итого	5.5	5.5	5.5	5.5	5.5	5.5	5.5	5.5	4.5	4.5	4.5	4.5	4.5	4.5	4.5	4.5	4.5	4.5	3.5	3.5

Рис. 2.17. Распределение работ для проекта Встреча

5.2. Зависимости

[Примечание для студентов. В этом разделе показываются взаимные зависимости различных работ. Данный раздел должен быть пересмотрен после проектирования. На данном этапе можно указать только зависимости самого верхнего уровня.]

Вторая итерация зависит от результатов первой. Инженер Фурнес занят в проекте Game 123, и с вероятностью 50 % он будет недоступен в первый месяц проекта.

5.3. Потребности в ресурсах

[Примечание для студентов. Оценка стоимости разработки первых трех версий игры Встреча приведена в разделе 8 этой главы. Вычисления и результаты можно поместить либо в данном разделе, либо в приложении.]

В проекте будут заняты семь инженеров, один секретарь на полставки и один технический специалист на полставки.

Аппаратные ресурсы составят восемь компьютеров Pentium 500 МГц с операционной системой Windows 95 и системой программирования Symantec Visual Café версии 3.0. Каждый компьютер должен иметь не менее 128 Мбайт RAM и не менее 6 Гбайт пространства на жестком диске.

5.4. Выделение бюджета и ресурсов

[Примечание для студентов. В этом разделе определяется, как расходуются выделенные средства. Смете предшествует оценка стоимости, разобранная в разделе 8 данной главы. По мере продвижения проекта оценка стоимости уточняется.]

Оценка до начала анализа требований.

Данная оценка проведена тремя различными методами.

1. С использованием неформальной оценки сверху вниз на основе предыдущего опыта команды по аналогичным проектам.
2. С использованием оценки сверху вниз на основе данных по отрасли.
3. С использованием оценки функционального размера для двух известных функций и экстраполяцией результатов на весь проект.

Результаты приведены в табл. 2.23 (все величины даны в тысячах строк исходного кода на языке Java).

[Примечание для студентов. Существует множество способов представления этих данных в зависимости от потребностей руководства и разработчиков. Некоторые из них даны в табл. 2.24. До анализа требований оценки остаются очень грубыми.]

Таблица 2.24. Предварительная оценка размера приложения до анализа требований

Метод	Минимум	Максимум	Комментарий
1	7,5	170	
2	4,2	300	
3	11,4	46	1,9–2,3 для двух функций, умноженные на 6–20 по числу функций всего приложения
Наиболее консервативная оценка	11,4	300	Максимум минимумов и максимум максимумов
Наименее консервативная оценка	4,2	46	Минимум минимумов и минимум максимумов
Наиболее широкий диапазон	4,2	300	Минимум минимумов и максимум максимумов
Наименее широкий диапазон	11,4	46	Максимум минимумов и минимум максимумов

Наиболее широкий диапазон используется потому, что оценки были получены при условии недостаточного взаимодействия с заказчиком.

- Оценка с использованием требований заказчика до начала сбора детальных требований.

  Должна быть представлена.

- Оценка с использованием детальных требований до начала проектирования архитектуры.

  Должна быть представлена.
- Оценка с использованием архитектуры до начала детального проектирования.

  Должна быть представлена.
- Оценка с использованием результатов детального проектирования до начала реализации.

  Должна быть представлена.
- Оценка в конце итерации 1 до начала итерации 2.

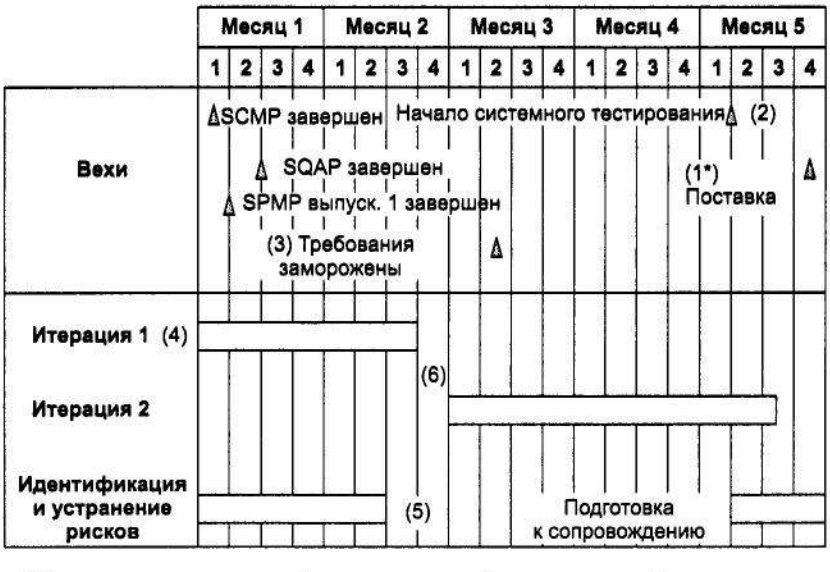
  Должна быть представлена.
- Оценка в конце итерации 2 до начала итерации 3.

  Должна быть представлена.

5.5. План-график

[Примечание для студентов. Если имеется фиксированный срок завершения и выбрана модель процесса, то этой информации достаточно для построения плана-графика верхнего уровня. По мере проектирования детальность плана-графика будет возрастать.]

План график приведен на рис. 2.18. См. также SQAP, где приведен план-график мероприятий по контролю качества.



**Номера показывают порядок, в котором создавалась эта таблица*

Рис. 2.18. Диаграмма задач для проекта Встреча при условии фиксированной даты поставки

6. Дополнения

6.1. Указатель

Должен быть представлен.

6.2. Приложения

Должен быть представлен.

Пример 2. План контроля качества (SQAP), часть 2

Первую часть SQAP см. пример 2 главы 1.

7. Тестирование

[Примечание для студентов. Здесь определяется, как будет осуществляться управление тестированием. Данный текст может ссылаться на STD, но не должен дублировать ее.]

На первых трех итерациях все функции по контролю качества выполняет ответственный за качество. На последующих итерациях отдел контроля качества должен выделить команду по контролю качества и передать ей эти функции. Описание команды по контролю качества будет представлено. Команда проекта *Встреча* несет ответственность за тестирование отдельных методов и комбинаций методов в классе (*модульное тестирование*). Ответственный за качество (в дальнейшем — команда по контролю качества) ответственен за все остальные виды тестирования (главы 8 и 9 данной книги). Дальнейшие детали тестирования изложены в STD проекта *Встреча*.

8. Отчеты о проблемах и коррекционная деятельность

[Примечание для студентов. В этом разделе объясняется, каким образом дефекты становятся известными, описываются и устраняются. Данный раздел не воспроизводит буквально детали стандарта IEEE. Читателю следует обратиться к стандарту IEEE или к книге Хэмфри [48] для получения более детальной информации о типах и классификации дефектов.]

Форма отчета об обнаруженном дефекте в программном обеспечении, которую должна использовать команда проекта *Встреча*, представлена на рис. 2.19. Для заполнения этой формы разработчики используют специальное приложение *describeDefect*. Номер дефекта определяется автоматически, и приложение гарантирует заполнение всех необходимых полей.

1. Номер дефекта:	2. Кто обнаружил:
3. Затрагивает документы:	
Затрагивает исходный код*	4. Пакет(ы) _____
5. Класс(ы) _____	6. Метод(ы) _____
7. Серьезность: _____	8. Тип: _____
9. Фаза появления**	
Треб ☐ Арх ☐ Дет. проект. ☐ Реализ. ☐ Интегр. ☐	
10. Детальное описание: _____	
11. Решение:	12. Статус открыт/закрыт:
13. Подписи: описание и план инспектированы: _____	
14. Код решения и план тестирования инспектированы: _____	
15. Изменение одобрено: _____	

*Для дефектов в исходном коде

**Самая ранняя фаза, на которой дефект уже существовал

Рис. 2.19. Форма отчета о дефекте

Различаются следующие значения уровня серьезности дефектов.

- Серьезный дефект: наличие такого дефекта влечет невыполнение требований к программному обеспечению.
- Тривиальный дефект: такой дефект не влияет на выполнение или сопровождение приложения.
- Незначительный дефект: дефект, который не относится к двум предыдущим категориям.

В случае обнаружения тривиального дефекта разработчик не обязан создавать отчет о дефекте — достаточно послать сообщение по электронной почте тому, кто наиболее вероятно может устранить этот дефект.

В документах встречаются дефекты следующих типов: отсутствие материала, неясность, неоднозначность, неполнота, повтор (в том же документе или в другом) и противоречие.

В исходном коде встречаются следующие типы дефектов: синтаксические, логические, ошибки в данных (то есть переменная принимает недопустимое значение) и нарушения режима безопасности (то есть возможен недопустимый обход правил безопасности).

Ответственный за качество (а в дальнейшем команда по контролю качества) создает и поддерживает базу данных дефектов, в которой собраны отчеты о дефектах, описывающих все проблемы, несоответствия и аномалии проекта *Встреча*. Ответственный за качество должен обеспечить систематическую фиксацию найденных дефектов по указанной форме, направление их на рассмотрение и устранение. Направление дефектов на рассмотрение должно производиться в соответствии с SCMP.

После третьей итерации при обнаружении дефекта ответственный за качество направляет отчет в Совет по контролю изменений (CCB — Change Control Board). На первых трех итерациях ответственный за конфигурацию будет выполнять функции команды по контролю качества, а лидер проекта будет выполнять функ-

ции Совета по контролю изменений в соответствии с SPMP. CCB оценивает полученный отчет о дефекте и назначает ему один из приоритетов: *Немедленно*, *Обязательно* или *Не обязательно*. Затем CCB назначает команду разработчиков проекта *Встреча*, команду по контролю качества или команду по управлению конфигурациями ответственной за решение проблемы. CCB также определяет срок решения проблемы, основываясь на приоритете проблемы и результатах анализа предшествующих отчетов о дефектах. После того как проблема, поставленная в отчете о дефекте, решена, команда по контролю качества инспектирует решение, а ответственный за качество информирует CCB о результатах инспектирования. Если необходимо, процесс повторяется.

9. Инструменты, технологии и методики

Методы управления качеством включают использование стандартов, прослеживание требований, верификацию проектирования, инспектирование программного обеспечения и верификацию формальными методами. Инструменты управления качеством включают в себя программы верификации программного обеспечения, списки контрольных вопросов, наклейки на носителях и штампы о приемке. Списки контрольных вопросов будут получены в лаборатории технологии программирования и адаптированы для проекта *Встреча* в соответствии с рекомендациями NASA [83]. В списки контрольных вопросов входят:

- списки контрольных вопросов для проведения совещаний, обзора документов и инспектирования;
- списки контрольных вопросов будут использоваться для верификации следующих видов деятельности и документов: предварительный обзор проектных решений (PDR), критический обзор проекта (CDR), экспертиза готовности к тестированию, аудит функциональной конфигурации, аудит физической конфигурации (PCA), SRS, SDD, SPMP, рабочие папки проекта (SDF);
- специальные формы и списки контрольных вопросов, используемые в целях проверки программного обеспечения.

[Примечание для студентов. В этой книге приведены несколько списков контрольных вопросов во врезках «Один из способов...». Например, эти списки контрольных вопросов охватывают процедуры проведения совещаний и инспектирования. Обычно команды начинают использовать опубликованные списки контрольных вопросов и постепенно приспосабливают их к нуждам своего проекта.]

Дополнительные инструменты, технологии и методы управления качеством описаны в SPMP.

10. Контроль программного кода

Методы и средства, используемые для поддержки, хранения и документирования версий компилируемого программного кода на всех фазах жизненного цикла, определены в SCMP. Команда по контролю качества должна убедиться, что процедуры, указанные в SCMP, действительно выполняются.

Наклейки на носителях с записанным программным кодом имеют особые отметки для верификации, дублирования и валидации. После завершения теста, при котором присутствовали и который проверили представители команды по контролю качества, ставится штамп «проверено командой по контролю качества» и ответственный представитель расписывается. До того как программный код будет использован, команда по контролю качества обязана убедиться, что используемая копия идентична проверенному оригиналу. Методы такой проверки включают в себя физическое наличие копий у персонала команды по контролю качества, проштампованные и подписанные наклейки на носителях, проверку контрольных сумм, побитовое сравнение и визуальное сравнение выходных данных для известных входных данных.

11. Контроль носителей

[Примечание для студентов. Здесь описываются правила обращения с дисками, лентами и т. п.]

Команда по контролю качества проверяет, что носители, используемые для управления конфигурациями, установлены и протестированы. Кроме того, команда по контролю качества проверяет то, что копирование носителей осуществляется только согласно процедурам, установленным в SCMP. Факт проверки носителя командой по контролю качества удостоверяется с помощью штампа на наклейке. По результатам проверки носителей представляется отчет.

12. Контроль поставщиков

[Примечание для студентов. Данный раздел регулирует взаимоотношения с поставщиками программного и аппаратного обеспечения. Здесь описывается, как и кто осуществляет эти отношения. Команда по контролю качества проверяет закупленные коммерческие программные продукты во время начальной проверки, удостоверившись, что имеются документы, подтверждающие целостность и версию продукта. Продукты валидируются в процессе установки и приемосдаточного тестирования.]

13. Сбор, сопровождение и хранение протоколов

[Примечание для студентов. Здесь определяется, как физически будут храниться записи и кто за них отвечает. Сюда же относятся файлы, которые не находятся под управлением конфигурациями.]

Записи, которые собирает и хранит команда по контролю качества, включают в себя:

- отчеты о выполненной работе;
- отчеты об обнаруженных аномалиях, которые составлены не по обычной форме отчетов о дефектах;

- памятные записки и рекомендации;
- книги учета работы команды по контролю качества;
- отчеты об аудитах;
- подписанные списки контрольных вопросов с ответами, полученными в ходе обзоров и аудитов;
- протоколы инспектирования.

Помимо проверки того, что процедуры архивирования, предписанные SCMP, выполняются, команда по контролю качества должна архивировать свои собственные материалы не реже раза в неделю. Эти записи должны храниться и на фазах сопровождения и эксплуатации.

14. Обучение

[Примечание для студентов. Обучение по контролю качества является специфичным для данного проекта.]

Для участников команды разработчиков должен быть организован вводный курс по качеству на 4 часа. Курс должен включать описание метрик, которые будут использоваться, и лабораторное занятие по изучению инструментов для сбора метрик. Далее, должны быть организованы ежемесячные трехчасовые занятия для того, чтобы поддерживать у разработчиков знания инструментов и методов контроля качества. Разработчики могут не присутствовать на этих занятиях, если они получают оценку «отлично», пройдя тест на сайте по адресу `GCI/monthly/SQA/quiz`.

15. Управление рисками

Команда контроля качества должна стараться обнаружить факторы риска как можно раньше. Процедуры управления рисками описаны в разделе 3.3 SPMP.

Глава 3
Анализ требований

Получение корректных требований — сложный процесс. Он состоит из чуткого взаимодействия с теми, кто финансово заинтересован в успехе данного программного приложения.

Этапы получения требований, обсуждаемые в этой главе, изображены на рис. 3.1.

- Разделы 3.1–3.7
- Руководство по учебному проекту: C-требования для учебного примера
- Упражнения
- Пример: Спецификация требований к программному обеспечению (SRS): часть 1 (часть 2 — конец главы 4)

Учебные цели этой главы.

- Отделить C-требования (требования заказчика) от D-требований (детальных требований).
- Освоить разные способы выражения C-требований:
  - варианты использования;
  - диаграммы переходов состояний;
  - диаграммы потоков данных;
  - наброски пользовательского интерфейса.
- Быть в состоянии написать первые части SRS.

3.1. Введение в анализ требований

В этой главе обсуждается общий анализ требований к приложению, что является процессом осмысления и последующего выражения концепций в конкретной форме. Большинство недостатков, найденных в произведенном программном обеспечении, возникло на стадии анализа требований. Практика показывает, что обычно такие недостатки труднее всего исправить.

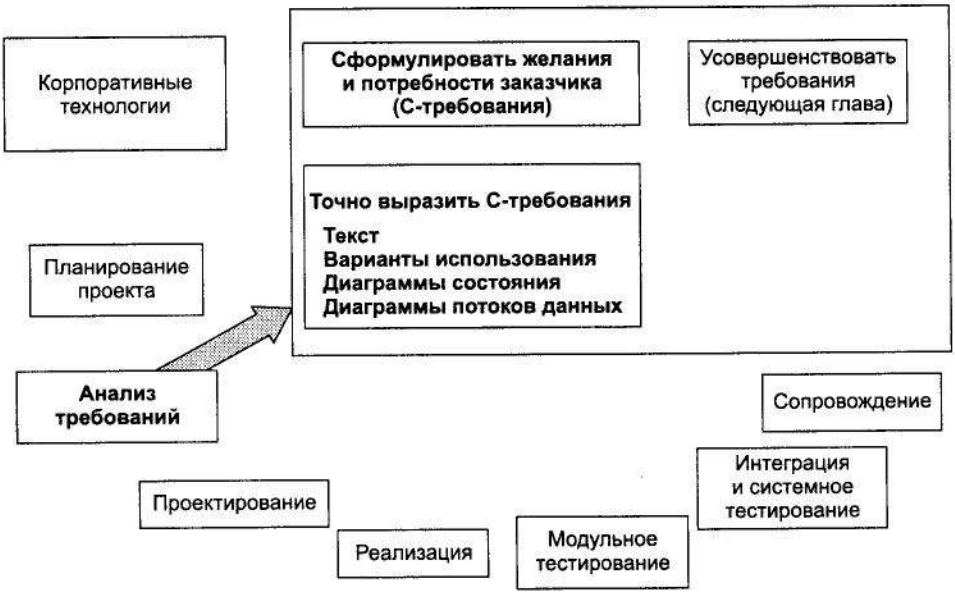


Рис. 3.1. Схема разработки программ: темы главы

3.1.1. Значение анализа требований

Чтобы построить что-либо, мы, прежде всего, должны понять, чем *это* должно быть. Процесс понимания и документирования *этого* и называется *анализом требований*. Обычно требования выражают, *что* приложение должно делать: зачастую здесь не пытаются сформулировать, *как* добиться выполнения этих функций. Например, следующее выражение (Y) является требованием для бухгалтерского приложения.

Система должна предоставлять пользователю доступ к балансу его банковского счета.

Вообще говоря, следующее выражение (N) не является требованием для приложения.

Балансы счетов клиентов будут храниться в таблице под названием «балансы» в базе данных Access.

Второе выражение касается того, *как* должно быть построено приложение, а не того, *что* это приложение должно делать.

Требование на одном уровне часто переходит в одно или несколько конкретных требований на следующем, более подробном уровне. Чтобы понять это, представьте себе, что ваше требование для будущего дома — «видеть горы на 180 градусов вокруг». Это требование может перейти в утверждение, что «терраса справа должна иметь размеры 20 × 50 футов». Это более конкретное требование на более детальном уровне. Аналогично, утверждение (*N*) действительно может стать требованием на последующих уровнях процесса разработки.

Более того, встречаются исключения из правила, запрещающего специфицировать в требованиях, как должно работать приложение. Например, у заказчика

могут быть особые причины потребовать, чтобы балансы счетов хранились в базе данных Access с конкретным именем, и тогда утверждение (*N*) действительно становится требованием.

Результатом анализа требований является документ, который обычно называют *спецификацией требований*, или *спецификацией требований к программному обеспечению* (SRS — Software Requirements Specification).

3.1.2. C-требования и D-требования

В течение некоторого времени проходили дебаты относительно того, кому «принадлежат» требования: заказчику или разработчикам (например, [10]). Для решения этого вопроса мы разделяем анализ требований на два уровня [93, 17]. Первый уровень документирует желания и потребности заказчика и пишется на языке, понятном заказчику. Результаты иногда называют *требованиями заказчика*, или *C-требованиями*. Первичной аудиторией для C-требований будет сообщество заказчиков, а уже вторичной — сообщество разработчиков. Второй уровень документирует требования в специальной, структурированной форме. Эти документы называются *требованиями разработчика*, или *D-требованиями*. Первичной аудиторией для D-требований будет сообщество разработчиков, а уже вторичной — сообщество заказчиков.

Данная книга использует стандарт IEEE для документирования требований. Разница между C- и D-требованиями согласно основным идеям шаблона документа стандарта IEEE проиллюстрирована на рис. 3.2. В данной главе обсуждаются C-требования. D-требования описаны в главе 4.

Рис. 3.2. Сравнение требований заказчика с детальными требованиями

Хотя целевые аудитории для C- и D-требований различны, заказчики и разработчики тесно сотрудничают при создании успешных продуктов. Один из способов, позволяющих обеспечить хорошее взаимодействие, — совместная работа представителей заказчика и разработчиков. Некоторые организации-разработчики

даже отказываются браться за работу без предоставления такой возможности. Это принцип *экстремального программирования*, упомянутого в главе 1. Участие представителей заказчиков в работе инженеров широко практикуется Министерством обороны США. Например, автор этой книги однажды работал в качестве сотрудника морского флота США над системой корабля вместе с инженерами-подрядчиками.

3.1.3. Почему требования следует написать

Даже новичку может показаться очевидным, что следует письменно формулировать, как должна вести себя завершенная программа. Однако этот процесс написания часто игнорируется или делается с ошибками. В таких случаях иногда считают, что исходный код выражает все требования: поскольку мы не можем обойтись без исходного кода, почему бы не свести весь процесс к этому единственному документу? Ответ на этот вопрос — так не пойдет. Теория разработки программ, самые опытные инженеры и эта книга — все настаивают на аккуратном документировании требований. Без таких документов команда практически не знает, каких целей она пытается достичь, не может корректно проверить свою работу, проследить свою производительность, получить адекватные данные по своей работе, предсказать объем и усилия в своей следующей работе и удовлетворить своих заказчиков. Короче говоря, не может быть профессиональной разработки без письменных требований.

Для иллюстрации этих моментов разберем следующее требование для научного приложения: *приложение должно показывать длину гена X12345 в системном окне (требование 7824)*.

Ниже приведен список действий, которые следует выполнить для этого и других требований. Каждое требование должно:

- быть четко выражено;
- быть легко доступно;
- быть пронумеровано;
- сопровождаться подтверждающими тестами;
- предусматриваться проектом;
- быть учтено кодом;
- быть протестировано отдельно;
- быть протестировано вкупе с другими требованиями;
- быть подтверждено тестированием после того, как выполнена сборка приложения.

Вдобавок, должно быть указано время, необходимое для выполнения каждого из этих шагов, чтобы в будущем можно было оценить время на осуществление похожих требований в аналогичных контекстах. Представьте себе, какой бы возник беспорядок, если бы требование 7824 *не было* записано явно. Немногие из упомянутых шагов могли бы быть выполнены корректно. Не удивительно, что приложение было бы ненадежно, не так ли? Когда вы обдумаете выполнение указанных шагов для каждого отдельного требования, у вас появится понимание

того, почему при разработке программ уделяется значительное внимание индивидуальным требованиям. Требования являются «универсальным мерилом» профессионализма.

3.1.4. Типичная схема процесса анализа требований

Типичная схема процесса анализа С-требований, описанного в этой главе, представлена на рис. 3.3. На каждой итерации мы будем заново смотреть на эту схему. На последнем шаге схемы предусматривается сбор детальных D-требований — процесс, описанный в следующей главе. Команда проводит измерения по стадиям процесса, чтобы дать возможность оценить будущие итерации и будущие приложения.

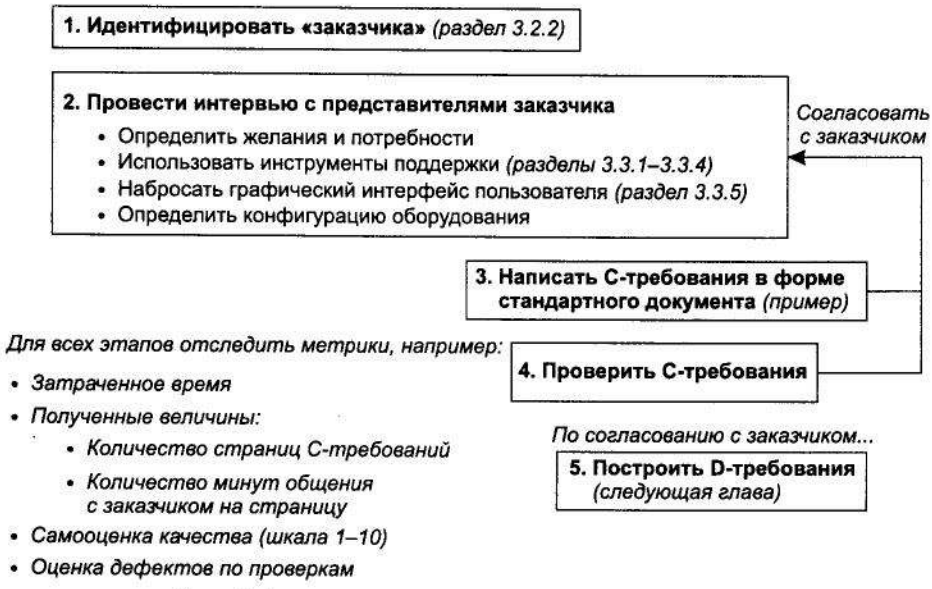


Рис. 3.3. Типичная схема для С-требований заказчика

Существует несколько способов организации SRS. Мы будем использовать — и модифицировать — стандарт IEEE 830-1993, оглавление которого приведено ниже. Содержание стандарта IEEE 830-1993 будет описано в этой главе. Раздел 3 стандарта, «Конкретные требования» (D-требования), будет расширен и использован в следующей главе.

1. **Введение**
   1.1. Цель
   1.2. Область применения
   1.3. Определения, термины и сокращения
   1.4. Ссылки
   1.5. Обзор

2. **Общее описание**
   - 2.1. Перспективы продукта
     - 2.1.1. Системные интерфейсы
     - 2.1.2. Пользовательские интерфейсы
     - 2.1.3. Аппаратные интерфейсы
     - 2.1.4. Программные интерфейсы
     - 2.1.5. Коммуникационные интерфейсы
     - 2.1.6. Ограничения по памяти
     - 2.1.7. Операции
     - 2.1.8. Требования по адаптации
   - 2.2. Функции продукта
   - 2.3. Пользовательские характеристики
   - 2.4. Ограничения
   - 2.5. Предположения и зависимости
   - 2.6. Распределение требований
3. **Конкретные требования (глава 4)**
4. **Сопровождающая информация (глава 4)**

Разработчики программного обеспечения спорят относительно качества разных форм документирования требований. Недостаток стандарта IEEE в том, что он относительно стар и обычно нуждается в некоторой модификации и дополнении (например, для отображения прогресса в объектно-ориентированном анализе и проектировании). Преимущество стандарта IEEE в том, что он однозначно решает большинство вопросов, которые можно было бы истолковать по-разному.

3.1.5. Преимущества анализа требований и проблемы, связанные с ним

Несовершенные требования (то есть не исправленные до полного завершения документа) обходятся очень дорого. По оценкам, их в 20–50 раз дороже исправить, если они попали в процесс разработки. В финансовых терминах, если стоимость обнаружения и исправления ошибки во время формулирования требований равна $100, то стоимость обнаружения и исправления того же самого дефекта в конце процесса разработки достигает от $2000 до $5000. Кто откажется вложить $100, если это будет гарантировать окупаемость от $2000 до $5000 через год или два? Думайте о поиске каждой ошибки на ранней стадии формулирования требований, как о таком вложении средств.

Убытки, связанные с отрицательным опытом работы заказчика с приложением, являются добавочным фактором при оценке расходов.

Почему же так много проектов понесло убытки от скудного или не проведенного анализа требований, если известна значительная прибыль от обнаружения ошибок во время формулирования? Основная причина заключается в том, что

заказчики обычно не знают в начале проекта, чего они хотят или в чем нуждаются. Изучение рынка видеоигр в конце главы является примером этой неуверенности: проект имеет сформулированную цель, но его содержание все еще меняется. В данной книге делается акцент на итерационную разработку и тщательную проверку требований, проектирования и реализации. Инженеры, использующие хорошо организованный итерационный процесс, собирают требования, проводят проектирование и выполняют реализацию согласованными итерациями.

Анализ требований — это необходимость, а не роскошь. Давайте убедимся в его эффективности на примере. Большинство организаций-разработчиков считают тестирование абсолютно необходимым. Но если бы кто-то дал вам просто черный ящик с одним фиолетовым, одним розовым и одним оранжевым колесиком и попросил вас протестировать этот ящик, вы бы, наверное, отказались. Тестирование было бы невозможным, если вы не знаете, что должно произойти! Другими словами, без требований продукт нельзя протестировать должным образом.

Многие организации не справляются с написанием требований. Это не значит, что они не пользуются требованиями — это только означает, что требования существуют в головах конкретных разработчиков приложения. После того как определены неэффективность незаписанных требований, наличие большого числа требований к каждому конкретному приложению и реалии текучести кадров, остаются ли еще вопросы относительно того, почему большая часть программных проектов никогда не доводится до конца? Более острая проблема создается организациями, которые пишут требования для начальной итерации, начинают по ним разработку, но не поддерживают изменения в документе с требованиями на последующих итерациях. Причина в том, что часто труднее обновить документ с требованиями, чем написать его первую версию. (Это подчеркивает важность хорошей организации документа с требованиями.) Тот факт, что обновление может быть более сложной задачей, однако, не отменяет банального утверждения о том, что неспособность сформулировать требования создаст очень много проблем.

Важным выигрышем от анализа требований является достижение понимания и согласия относительно приложения, которое вы собираетесь создать [17]. Это базис контрактов любого вида.

Большинству из нас говорили, что «писать — значит думать», и частично это справедливо при формулировании требований. Довольно много разработчиков пытаются избежать формулирования требований, нетерпеливо приступая прямо к написанию кода. По опыту автора книги один разработчик, отрицательно относящийся к процессу, вдруг начинает производить поразительное количество кода. Затем он объявляет команде: «Я выполнил свою часть — теперь вы, ребята, можете тратить свое время на составление всех сопровождающих документов. А я пошел». Такое кодирование сравнимо с выливанием тонн бетона для постройки моста, не зная, откуда и куда он должен в итоге быть построен (не говоря уже о его финальном дизайне). Автор считает, что нежелание писать требования связано не с тем, что написание считается слишком простым, чтобы о нем беспокоиться, а с тем, что корректное написание требований на самом деле довольно сложно. Конкретные, реалистичные и измеряемые процедуры для написания требований являются профессиональным испытанием.

3.2. Взаимодействие с заказчиком

3.2.1. Источники возникновения требований

Этот раздел сконцентрирован на вопросах взаимодействия с людьми для анализа требований. Именно люди, бесспорно, являются источниками возникновения требований. Брэкет [17] отобразил на графике приложения нескольких типов (рис. 3.4), чтобы продемонстрировать уровень, до которого требования собираются у людей, а не из других источников, таких как письменный материал. График классифицирует приложения по степени *наложенных на них ограничений*. Это касается требований к приложениям, которые нельзя изменить. Например, на приложение, описывающее траекторию полета ракеты, наложены ограничения силы тяжести, на химические реакции — ограничения законов физики. Чем меньше объективных ограничений имеет проблема, тем большее количество ее ограничений должно быть получено от людей. Например, в экстремальном случае — нашей видеоигре, поскольку это продукт полностью воображаемый, большинство ограничений должно быть получено от людей. В противоположном случае — системы управления ракетами ограничены скорее физикой движения, а не пожеланиями людей, и поэтому требования берутся из физических уравнений.

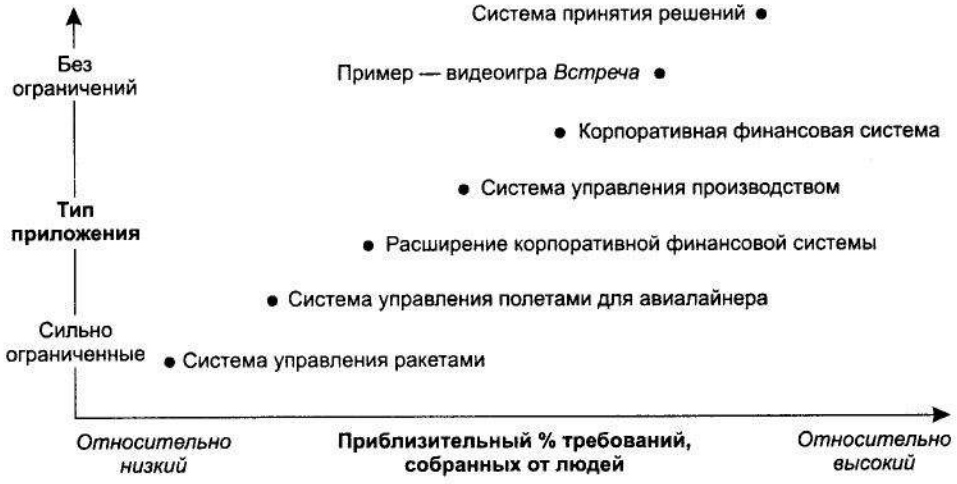


Рис. 3.4. Источники возникновения требований: люди и другие источники

3.2.2. Определение заинтересованных лиц

Люди, имеющие долю в результирующем продукте, называются людьми, финансово заинтересованными в проекте. Вообще говоря, они являются заказчиками приложения. Как пример, представьте себе создание сайта для электронной коммерции. Один набор финансово заинтересованных лиц состоит из посетителей сайта: их типичное первое требование — легкость, с которой они могут найти и купить необходимые товары. Владельцы компании также финансово заинтере-

сованные лица: их основным требованием может быть прибыльность, краткосрочная или долгосрочная. По этой причине они, возможно, захотят, чтобы на сайте был сделан акцент на дорогих товарах. Менеджеры — другая группа финансово заинтересованных лиц — могут потребовать, чтобы приложение вело учет посетителей. Разработчики приложения также являются финансово заинтересованными: они могут захотеть рискнуть использовать новую технологию, чтобы идти в ногу со временем.

В случае готовых к употреблению («коробочных») приложений, таких как системы обработки текстов, электронные таблицы и среды разработки, разработчики уделяют основное внимание доступности приложения как можно большему числу пользователей. Хотя это может быть сложной рыночной проблемой, очевидно, что пользователи являются наиболее значительными финансово заинтересованными лицами. Для большинства крупных проектов, однако, определение наиболее важных финансово заинтересованных лиц является сложной проблемой. Заказчик часто является стороной, оплачивающей разработку приложения, но даже это не всегда корректно. Например, морской флот может платить за разработку, но каждодневными заказчиками разработчиков могут быть государственные служащие, а не морские офицеры. Но тогда разве не являются налогоплательщики заказчиками, поскольку на самом деле это они платят за приложение? Заказчиком субподрядчика является основной подрядчик. Заказчиком коробочного приложения — множество потенциальных покупателей, установленное отделом маркетинга. Когда приложение разрабатывается для внутреннего использования в компании, такого как обработка заявок в страховой фирме, заказчиком является сама компания.

Конфликтующие интересы финансово заинтересованных лиц могут легко привести к противоречивым требованиям. Пример этого — когда две разных группы в компании по разным мотивам заказывают «одно и то же» приложение. Как результат, требования могут быть не согласованными. Когда требования нельзя согласовать, проекты имеют тенденцию продвигаться с трудом и часто прекращаются. Даже когда требования финансово заинтересованных лиц согласованы, они могут быть слишком дорогими, чтобы их полностью удовлетворить.

Разработчики несут профессиональную ответственность, что может основательно влиять на требования. Предположим, например, что разработчиков попросили создать программное обеспечение для медицинского устройства, причем бюджет разработки фиксирован, однако разработчики установили, что требуемые характеристики невозможно в достаточной мере протестировать, не выходя за пределы бюджета. Если бюджет не будет изменен, им придется поступиться требованиями. Даже если от этого не будет зависеть человеческая жизнь, разработчики в любом случае несут социальную ответственность за производство продуктов, в точности соответствующих предъявляемым к ним требованиям. Это в большой мере затрагивает управление требованиями проекта, с тем чтобы сформулированные требования можно было выполнить в пределах бюджетных или временных ограничений.

Хороший руководитель проекта умеет преодолевать эти трудности — это процесс, требующий организаторских, личных, деловых и политических навыков.

3.2.3. Примеры пожеланий заказчиков

Когда группа разработчиков начинает анализировать требования, заказчик обычно все еще формирует концепции того, что он хочет и что ему нужно. Это аналогично этапу уточнения требований при взаимодействии архитектора и клиента. Например, клиент может захотеть ранчо с четырьмя спальнями и большой гостиной. Но при уточнении требований клиент вынужден полагаться на архитектора, который должен помочь клиенту понять, чего последний хочет (например, просторную гостиную, в которой можно разместить 10 человек).

Пример, используемый в этой книге, — это видеоигра *Встреча*. Ниже приведен фрагмент мнения заказчика, полученного вымышленным коммерческим отделом.

Встреча — это ролевая игра, моделирующая всю жизнь персонажа игрока или ее часть. Она должна быть интересна как мужчинам, так и женщинам.

Приведем результат С-требований для этого примера.

- Это ролевая игра, моделирующая все стороны жизни персонажа игрока.
- Игровые персонажи, не контролируемые игроком, называются *внешними персонажами*.
- У каждого игрового персонажа есть набор *характеристик*, таких как *сила*, *скорость*, *терпение*.
- Каждая характеристика имеет численное значение.
- Персонажи *встречаются*, если находятся в одной зоне, и могут затем *вступать в контакт* друг с другом.
- Результат контакта зависит от значений характеристик и от зоны, в которой произошел контакт.
- Персонажи игрока могут перераспределять свои характеристики в отсутствие внешнего персонажа.
- Перераспределение вступает в силу с некоторой задержкой, во время которой игрока могут принудить к контакту.
- Успех определяется по одному из принципов:
  - максимальное *число очков-жизней*, накопленное игроком;
  - выживание в течение как можно более продолжительного периода.

Полный текст отчета приведен в разделе «Общее описание» учебного примера. Примерами нерешенных вопросов являются: один или несколько персонажей игры будут находиться под контролем игрока; что конкретно происходит при общении двух персонажей; можно ли будет играть в эту игру через Интернет. Задачей инженера является выяснение этих вопросов с заказчиком.

Нужды заказчика несколько труднее определить, чем его *желания*. Например, заказчик может *захотеть*, чтобы музыкальное приложение позволяло компьютерным новичкам гарантированно записывать музыку, но ему также может быть *нужна* периодическая функция автосохранения для избежания потери работы.

Является ли последняя характеристика требованием, или это часть проектирования? Это зависит от соглашения между разработчиком и заказчиком. Если заказчик, осознав смысл автосохранения, принимает решение, что он действительно хочет иметь такую функцию, автосохранение становится требованием. Заказчик может согласиться, однако, оставить разработчику право решать, как добиться безопасной работы для новичков. В этом случае автосохранение будет не требованием, а элементом дизайна, содействующим удовлетворению требований.

3.2.4. Проведение опроса и документирование

Большая часть анализа требований является коммуникационной деятельностью, тщательно организованной для получения наилучших результатов.

ОДИН ИЗ СПОСОБОВ ПРОВЕДЕНИЯ ОПРОСА ЗАКАЗЧИКА

Перед интервью:

1. Перечислите и расставьте приоритеты в списке интервьюируемых.
2. Спланируйте интервью с фиксированным временем начала и конца:
   - должны присутствовать как минимум два члена команды разработчиков;
   - приготовьтесь записывать на пленку.

Во время интервью:

3. Сконцентрируйтесь на слушании;

   не будьте пассивны: исследуйте сами и побуждайте собеседника:
   - настаивайте на понимании *желаний* и изучении *нужд*;
   - обсудите варианты использования, а также потоки данных и диаграммы переходов состояний;

   делайте подробные заметки.
4. Спланируйте следующую встречу.

После интервью:

5. Составьте черновик С-требований SRS, используя стандарт.
6. Свяжитесь по электронной почте с заказчиком для получения его замечаний.

Поскольку обычно имеется несколько финансово заинтересованных лиц, желающих сделать свой вклад в общее дело, первый вопрос — решить, кого опрашивать. Вместо того чтобы пытаться предоставить каждому одинаковое количество времени, что может привести к противоречивым требованиям и зря потраченным усилиям, автор рекомендует выбрать одно или, возможно, два основных заинтересованных лица, опросить их, а затем уже собрать комментарии у остальных основных финансово заинтересованных лиц. Процесс опроса мнений заказчиков обычно дорогостоящий, требующий значительных временных затрат более чем одного человека. По этой причине его следует точно спланировать. Предпочтительнее иметь двух интервьюеров на каждом собрании, поскольку считается, что один интервьюер имеет тенденцию пропускать некоторые важные вопросы. Может также оказаться полезным записать собеседование на пленку, но не забудьте заранее спросить разрешения у всех участников.

Хотя очень важно внимательно слушать заказчика во время интервью, обычно бывает трудно сформулировать требования, слушая заказчика в одиночку. Часто заказчик формулирует требования по ходу разговора и нуждается при этом в помощи. Хотя в основном концепция формируется со слов заказчика, интервьюер и заказчик разрабатывают концепцию до некоторой степени совместно. Заказчику часто бывают необходимы подсказки для завершения формирования концепции; в чем-то это напоминает (хотя и не очень сильно) выступление свидетеля в суде.

Мы придаем особое значение использованию примеров как эффективному способу получить и сформулировать требования для широкого спектра приложений. Для некоторых требований необходимо составить диаграммы; в разделах 3.3 и 3.4 описаны разные способы. Для утверждения требований в письменной форме интервьюеры подготавливают и посылают по электронной почте список требований, при необходимости устраивая повторное собрание. Не забывайте также, что еще придется формулировать D-требования, что также потребует времени на проведение собраний.

После собрания делается черновик C-требований, например, в формате стандарта IEEE. Затем этот черновик посылается по электронной почте заказчикам для комментариев. Повторные опросы проводятся до полной удовлетворенности заказчиков C-требованиями.

3.3. Описание C-требований (требований заказчика)

Нам предстоит сложное испытание, когда придется четко сформулировать, что хотят заказчики и что им нужно. В некоторых случаях одних слов может оказаться достаточно, но для большого количества приложений описательный текст должен сопровождаться разнообразными количественными оценками. В разделе 3.6 подведен итог различным методикам формулирования требований.

3.3.1. Концепция работы

Заказчики разрабатывают концепцию — часто подсознательную и неполную — того, как их приложение будет работать. Эту концепцию иногда называют *моделью* приложения, или *концепцией работы*. Разные люди обычно имеют различные представления о программном продукте. Например, возможным представлением о работе бюро погоды может быть одно из нижеследующего.

- Средство для преобразования необработанных материалов метеоцентра в графическое представление.
- Система реального времени для предсказания погоды.
- Приложение, предупреждающее пользователей о погодных аномалиях.

Эти различные представления о работе программного продукта приведут к совершенно разным приложениям.

Менеджер проекта или разработчик требований помогает заказчику четко сформулировать его концепцию работы. Поскольку обычно заказчики не владеют технологией выражения таких концепций, инженеры могут предложить подходящие технологии, такие как *варианты использования*, *потоки данных* или *переходы состояний*, описанные ниже. Обратите внимание, что эти техники также используются и в проектировании, как показано в главах 5 и 6.

3.3.2. Варианты использования

Требования часто естественно выразить через взаимодействие приложения с внешним пользователем. *Варианты использования*, концепция, которую изобрел Якобсон [63], является очень полезным способом выражения требований заказчика в форме таких взаимодействий. Вариант использования определяется, прежде всего, своим именем и типом пользователя приложения, называемого *действующим лицом*.

Вариант использования состоит из типичного взаимодействия между действующим лицом и приложением. Например, команда *открыть файл* будет типичным вариантом использования текстового редактора с пользователем в качестве действующего лица, что, в свою очередь, может состоять из следующей последовательности шагов.

1. *Пользователь* открывает меню Файл.
2. *Система* показывает команды Новый и Открыть.
3. *Пользователь* выбирает Открыть.
4. *Система* показывает окно файлов.
5. *Пользователь* вводит каталог и имя файла.
6. *Пользователь* щелкает на кнопке Открыть.
7. *Система* находит упомянутый файл и открывает его в окне текстового редактора.

Поскольку описание вариантов использования легко понять, варианты использования являются особо полезным способом коммуникации с заказчиком. Мы даже вынесем идею вариантов использования за пределы С-требований.

Один и тот же человек может использовать систему несколькими разными способами, принимая роли разных *действующих лиц*. Примеры четырех случаев использования для примера видеоигры изображены на рис. 3.5 и 3.6 (два из них расписаны подробно).

В нотации UML овал означает вариант использования. В двух случаях действующим лицом является играющий. Каждый вариант использования является последовательностью действий, предпринимаемых игроком и игрой *Встреча*, как показано в варианте использования «Инициализировать». Вариант использования «Вступить в контакт с внешним персонажем» представляет собой типичную последовательность действий игры и игрока каждый раз, когда главный персонаж игрока и внешний персонаж оказываются одновременно в одном месте. В варианте

использования «Установить правила» действующим лицом является разработчик игры: *действующее лицо* описывает возможность игры поддерживать редактирование правил взаимодействия персонажей. Обратите внимание, что вариант использования «Переместиться в соседнюю зону» исследуется в примере в конце главы, а вариант использования «Установить правила» не включен в пример.

Рис. 3.5. Вариант использования «Инициализировать» в игре Встреча

Рис. 3.6. Вариант использования «Вступить в контакт с внешним персонажем» в игре Встреча

Действующее лицо не обязательно должно быть человеком: это может быть другая система, использующая приложение. Например, если разрабатываемое при-

ложение является системой управления робота, действующим лицом может быть автоматизированная система завода, использующая систему управления роботом.

Варианты использования могут описывать лишь ограниченное ветвление, если имеется более одного уровня ветвления, вариант использования, вероятно, следует разбить на несколько. Даже одиночное ветвление в варианте использования приводит к неловким описаниям. Например, приведенный ниже текст может быть вариантом использования приложения для ведения личного бюджета.

1. Пользователь выбирает «добавить чеки» или «согласовать счет».
2. Если выбрано «добавить чеки»:
3. Происходит одно действие.
4. Происходит другое действие.
5. ...
6. Если выбрано «согласовать счет»:
7. Происходит одно действие.
8. Происходит другое действие.
9. ...

Это было бы лучше разбить на варианты использования «Выбрать вариант», «Добавить чеки» и «Согласовать счет».

Поскольку варианты использования аналогичны рассказам, они эффективны для извлечения информации из заказчиков и предоставляют превосходное понимание приложения. Варианты использования можно формулировать с разным уровнем обобщения. USDP [64] рекомендует использовать подробные варианты для определения большинства требований.

Очевидно, что похожие варианты использования не имеют большой ценности. Один относительно другого, варианты использования должны быть либо *последовательными*, либо *ортогональными*. Два варианта использования являются последовательными, если один непосредственно следует за другим. Ортогональные варианты использования относятся к абсолютно разным возможностям. Например, в приложении для склада товаров варианты использования для действующих лиц *начальник* и *финансовый аналитик* будут ортогональными. В примере видеоигры *Встреча* вариант использования «Установить правила» является ортогональным для варианта использования «Встретить внешний персонаж». В главе 4 показано, как варианты использования комбинируются для создания новых вариантов; там также представлено наследование вариантов использования.

Якобсон [63] дал начало идее о вариантах использования, заметив, что несмотря на колоссальное количество потенциальных вариантов выполнения, в большинстве приложений было задумано относительно немного типичных вариантов использования. Он предложил начинать проектирование приложения с написания вариантов использования с последующим их применением для выбора классов. Эта техника продемонстрирована в главе 4. Варианты использования также могут применяться для разработки прототипа (см. далее). Они являются основой для тестовых планов системного уровня. Мы продемонстрируем эти стратегии на нашем примере.

Большинство установленных стандартов документации, в том числе IEEE, появились раньше идеи вариантов использования и должны быть расширены для согласования с ними. Варианты использования полезны при формулировании требований, проектировании и тестировании. Поскольку высокоуровневые варианты использования могут выражать концепцию заказчика относительно работы приложения, наш пример включает их в раздел «общего представления» SRS.

Хотя варианты использования часто отождествляются с объектно-ориентированными методами, они могут также использоваться с любой методологией разработки.

3.3.3. Диаграммы потоков данных для общения с заказчиком

Некоторые требования естественно описаны потоками данных между обрабатывающими элементами. В *диаграмме потоков данных* узлы, отмеченные кругами или прямоугольниками, представляют обрабатывающие единицы. Стрелки между ними, подписанные типами данных, обозначают потоки данных. Хранилища данных — места, где хранятся данные, такие как базы данных — отмечены горизонтальными линиями с именем хранилища между ними. Внешние объекты, такие как пользователь и принтеры, представлены прямоугольниками.

Предположим, например, что наш заказчик пытается объяснить тип банковского приложения, который ему нужен, начиная с депозитов на счет. Функциями депозита могут быть *получить депозит от пользователя, проверить операцию по депозиту, чтобы убедиться в ее законности*. Эти функции представлены кругами на рис. 3.7. Далее, тип данных, передаваемых между этими функциями, отмечен на рисунке — номер счета и размер депозита. Пользователь также участвует, и это должно быть отмечено на диаграмме. В качестве другого примера рассмотрим функцию для подведения баланса счета, которая требует входные данные из хранилища данных, как показано на рисунке. Более полная диаграмма потоков данных для банковских требований показана на рис. 3.8.

Полная диаграмма содержит хранилище данных («Банки»), представляющее собой список банков, предоставляющих депозит через этот банкомат. Здесь также показан поток данных в ответ на запрос с более подробными данными счета.

Формулирование требований в текстовой форме только усложнило бы задачу по сравнению с использованием диаграммы потоков данных. Обратите внимание, что диаграммы потоков данных, однако, не показывают управление. Например, приложение банкомата не показывает, какая функция выполняется первой.

Стандарты, используемые для символов, различаются: например, для обрабатывающих элементов часто используются прямоугольники вместо кругов.

От приложения зависит, помогут диаграммы потоков данных выразить требования или нет. Например, диаграмма потоков банковских данных на рис. 3.8 разъясняет многим читателям, как приложение должно себя вести, в то время как требования к видеоигре, вероятно, не были бы внятно объяснены с помощью такой диаграммы. Диаграммы потоков данных широко используются для описания проектных решений, и мы повторно обратимся к ним в главах 5 и 6.

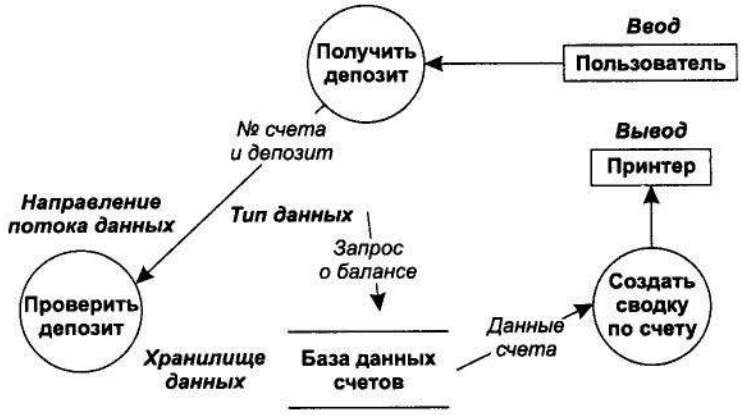


Рис. 3.7. Диаграмма потоков данных: объяснение символов

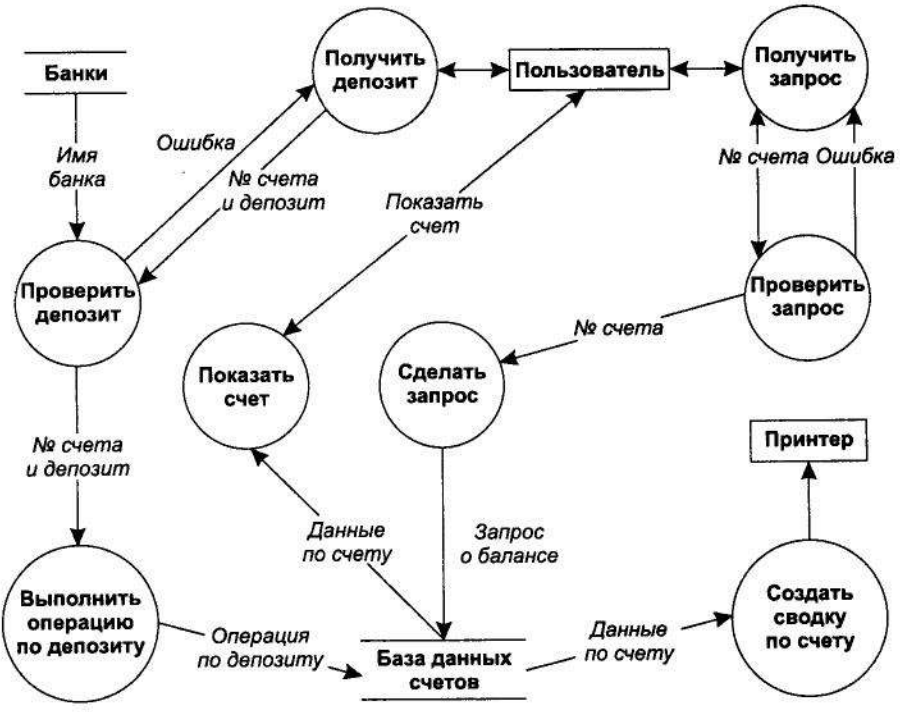


Рис. 3.8. Неполная диаграмма потоков данных для программы управления банкоматом

3.3.4. Диаграммы переходов состояний

Иногда приложение или его часть лучше всего представлять себе в одном из нескольких состояний. *Состоянием* приложения является его статус или ситуация, в которой оно находится. Состояния иногда называют *фазами* или *стадиями*. Идея заключается в разбиении приложения на состояния, так чтобы приложение

всегда находилось в одном и том же состоянии. Например, может быть полезно представить себе книжный интернет-магазин, который постоянно находится либо в состоянии *просмотра* (изучение информации о книгах), либо в состоянии *покупки* (с предоставлением информации о кредитной карте и т. д.). Концепция состояния имеет четкое определение в контексте реализации, но в настоящем контексте определение все еще неформальное.

Существует несколько возможных состояний игры *Встреча*.

- *Настройка*: состояние, в котором можно производить настройку игры.
- *Подготовка*: снаряжение персонажа игрока такими характеристиками, как сила и ум, может выполняться в отсутствие внешних персонажей.
- *Ожидание*: в этом состоянии с игроком ничего не происходит.
- *Контакт*: состояние, когда сравниваются значения характеристик персонажа игрока и внешнего персонажа.

Эти состояния изображены на рис. 3.9. Для приложений, управляемых событиями, диаграммы переходов состояний иногда могут быть эффективным способом достижения соглашения разработчика и заказчика относительно того, как приложение должно работать.

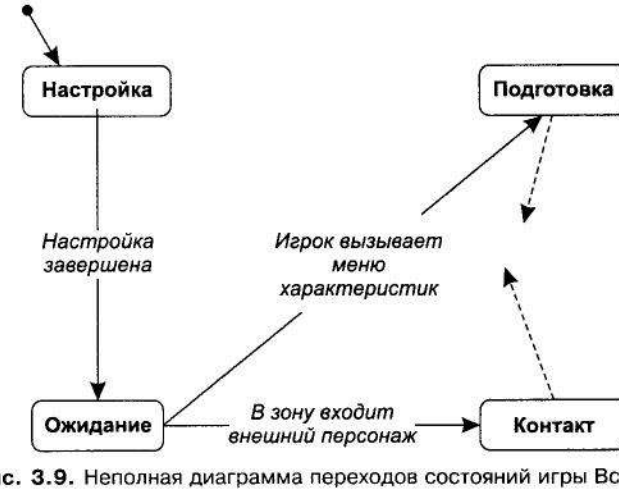


Рис. 3.9. Неполная диаграмма переходов состояний игры Встреча

После определения состояний добавляются переходы между ними. Переходы обозначаются стрелками, каждая из которых помечена именем события, приводящего к смене состояния приложения. Иногда, когда объект находится в заданном состоянии, при возникновении события объект может перейти в одно из нескольких состояний в зависимости от *условия*. Например, когда игрок решает переместить свой персонаж, игра переходит из состояния *Ожидание* в одно из двух состояний: одна возможность — вернуться обратно в состояние ожидания (если в новой зоне, куда попал персонаж игрока, нет внешних персонажей); другая возможность — переход в состояние *Контакт* (если в новой зоне присут-

ствует внешний персонаж). В UML условия записываются в квадратных скобках (рис. 3.10).

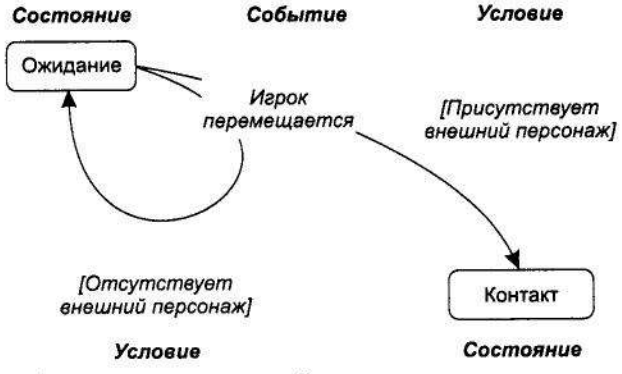


Рис. 3.10. Использование условий в диаграммах переходов состояний

Полная диаграмма переходов состояний для игры *Встреча* изображена на рис. 3.11. Как только игрок закончил настройку игры, происходит переход из состояния *Настройка* в состояние *Ожидание*. Если игра находится в состоянии *Ожидание* в момент появления внешнего персонажа, осуществляется переход в состояние *Контакт*. Обратите внимание, что процесс настройки характеристик и процесс передачи результатов могут быть нарушены появлением внешнего персонажа, что немедленно приводит к возникновению новой встречи.

Рис. 3.11. Диаграмма переходов состояний для игры Встреча

Модель переходов состояний — хороший способ объяснить концепцию работы игры *Встреча*. Модели переходов состояний широко используются и как инструмент проектирования (главы 5 и 6). Стоит ли использовать модели переходов состояний для формулирования С-требований, как мы это здесь сделали, зависит от конкретного приложения и от того, насколько это может помочь заказчику. Обычно для этого требуется наличие у заказчика некоторых навыков.

3.3.5. Черновик пользовательского интерфейса и других интерфейсов

Дизайн пользовательского интерфейса входит в фазу проектирования программного обеспечения, однако его также можно считать и частью фазы требований. Это лишь вопрос предпочтения. В этой книге рассматривается вторая альтернатива, включающая в себя только проектирование *программы* на фазе проектирования.

Заказчики часто представляют себе приложение в форме графического пользовательского интерфейса, и хорошим способом помочь им описать программу является разработка чернового варианта пользовательского интерфейса. В частности, в главе 4, в примере, приводится спецификация пользовательских интерфейсов в контексте подробных требований. Нашей целью является представление важнейших элементов проектирования интерфейса. Это довольно сильно отличается от *технического* проектирования программы, описанного в главах 5 и 6.

В разработке пользовательских интерфейсов для наших прикладных задач наиболее удачливые из нас имеют возможность либо работать с профессиональным дизайнером, либо хотя бы иметь его поддержку. Однако для многих проектов, особенно небольших, разработчики программного обеспечения должны разрабатывать дизайн пользовательского интерфейса сами. Поэтому мы перечислим некоторые основные принципы проектирования пользовательского интерфейса.

3.3.5.1. Шаги разработки пользовательских интерфейсов

В [35] предлагается 11 этапов разработки пользовательских интерфейсов. Автор упростил их; каждый из этих шагов применим к процессу обработки требований заказчика и (или) процессам обработки подробных требований.

1. Узнайте своего пользователя (С) (обработка С-требований).
2. Поймите назначение проектируемой системы (С).
3. Примените принципы хорошего экранного дизайна (С, D).
4. Подберите подходящий тип окон (С, D).
5. Разработайте системные меню (С, D).
6. Выберите соответствующие аппаратные устройства управления (С).
7. Выберите соответствующие экранные элементы управления (С).
8. Организуйте и создайте раскладку окон (С, D).

9. Выберите подходящие цвета (D).
10. Создайте осмысленные значки (C, D).
11. Предоставьте эффективные сообщения, обратную связь и руководство (D).

Шаг 1 (знакомство с пользователем). На этом шаге рекомендуется оценить общество конечных пользователей программы. В общих чертах основные факторы намечены в табл. 3.1.

Таблица 3.1. Критерии, по которым оцениваются потенциальные пользователи программы

Характеристика	Градации
Уровень знаний и опыт	
Компьютерная грамотность	Высокий
	Средний
	Низкий → объяснить каждый термин
Системный опыт	Высокий
	Средний
	Низкий → предоставить примеры и анимацию
Опыт работы с подобными программами	Высокий
	Средний
	Низкий → предоставить примеры и анимацию
Образование	Ученая степень
	Колледж
	Школа → использовать термины 12-го класса
Уровень чтения	>12 лет в школе/5–12/
	<5 → использовать очень простой язык
Машинопись	135 слов в минуту / 55 / [10 → предоставить небольшие поля для ввода текста, примеры, уделить особое внимание формам для заполнения]
Физические характеристики пользователя	
Возраст	Молодой / среднего возраста / пожилой
Пол	Мужской / женский
Развитость рук	Левша / правша / владеющий одинаково обеими руками
Физические недостатки	Слепой / дефекты зрения / глухой / моторные недостатки
Характеристики заданий и работы пользователя	
Способ использования этой программы	По усмотрению / [обязательная → сделайте программу интересной в использовании]
Частота использования	Постоянная / частая / случайная / [разовая → предоставить всю справочную информацию с каждым экраном]
Коэффициент текучести кадров	Низкий / средний / [высокий → предоставить всю справочную информацию с каждым экраном]
Важность задания	Высокая / средняя / [низкая → сделать интересной в использовании]

продолжение

Таблица 3.1 (*продолжение*)

Характеристика	Градации
Повторяемость задания	Низкая / средняя / [высокая → автоматизировать как можно больше шагов, предоставить разнообразие в представляемых данных, предоставить возможности обучения]
Предварительное обучение	Нет / самостоятельное изучение по справочникам / [интенсивное → предоставить интерактивную систему обучения]
Категория работы	Администратор / менеджер / профессионал / секретарь / [клерк и т. д. → использовать язык, примеры и описания, знакомые обычному клерку]
Психологические характеристики пользователя	
Вероятное отношение к работе	Положительное / безразличное / отрицательное
Вероятные мотивации	Высокие / средние / [низкие → сделать приложение особенно привлекательным]
Стиль процесса познания	Словесный или [пространственный → подчеркнуть геометрический вид]
	Аналитический или [интуитивный → подчеркнуть символы в тексте]
	Конкретный или [абстрактный → разработать обобщения]

Список контрольных вопросов — это способ убедиться в том, что мы знаем основные характеристики ожидаемых пользователей и что мы документируем наши предположения. Позднее эти характеристики определят природу пользовательского интерфейса. В общем случае, плохо образованные пользователи с меньшим объемом навыков и мотиваций требуют гораздо большей простоты, более подробных объяснений и больше помощи. Возможно, это будет сделано в ущерб эффективности и скорости. Часто желательно предоставить несколько уровней пользовательского интерфейса в зависимости от уровня знаний пользователя.

Шаг 2 (понимание назначения). На этом шаге от дизайнера требуется понимать цель конкретного предлагаемого пользовательского интерфейса в свете общего назначения программы. Например, если целью является инвентаризация склада, мы можем захотеть, чтобы пользовательский интерфейс отражал план склада. Последовательность экранов может отражать способ, которым обычно пользователи выполняют свои задания вручную.

Шаг 3 (понимание принципов хорошего экранного дизайна). Перечислим некоторые основные элементы хорошего экранного дизайна.

1) Убедитесь в единообразии экранов приложения, а также в логичности каждого отдельно.
   - Соглашения; процедуры; местоположение.

2) Сделайте предположение о том, откуда обычно пользователь будет начинать работу.
   - Часто «первый» элемент размещают в верхнем левом углу.
3) Сделайте навигацию как можно более простой:
   - выровняйте похожие элементы;
   - сгруппируйте похожие элементы;
   - учтите границы вокруг похожих элементов.
4) Примените иерархию для подчеркивания порядка важности.
5) Примените принципы приятных визуальных эффектов:
   - баланс, симметрия, регулярность, предсказуемость;
   - простота, единообразие, пропорциональность, экономия.
6) Предоставьте подписи.

Здесь также отмечены несколько факторов, которые часто применяются для создания «приятного» интерфейса. Хотя это не более чем введение в область визуальных эффектов, данные принципы довольно полезны даже на этом уровне. В качестве примера мы применим некоторые из этих принципов к экрану, используемому для ввода информации о заказчиках и их счетах. Для улучшения интерфейса мы начнем с левого верхнего угла, размещая в первую очередь наиболее важные элементы и группируя похожие элементы. Улучшение программы, достигнутое в результате применения этих принципов [35], представлено на рис. 3.12 и 3.13. На рис. 3.14 показано, где использовались некоторые принципы хорошего экранного дизайна.

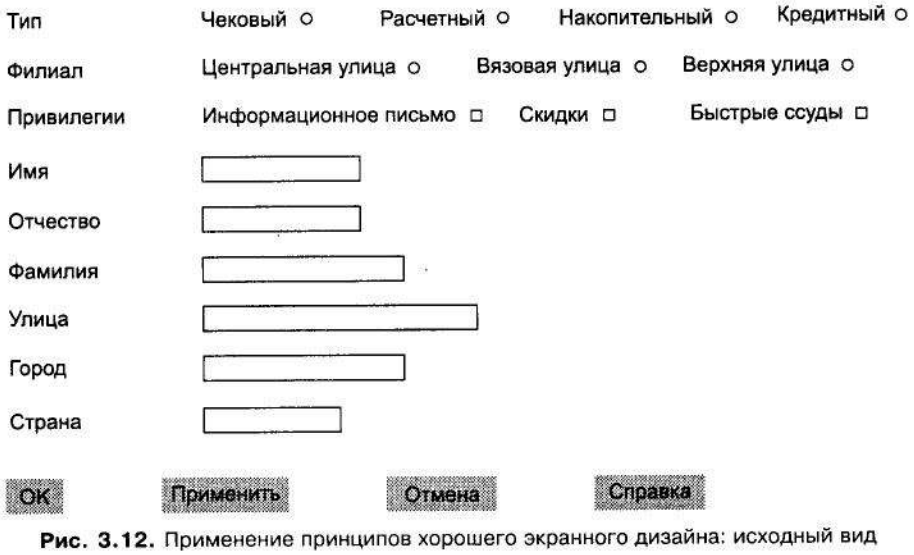


Рис. 3.12. Применение принципов хорошего экранного дизайна: исходный вид

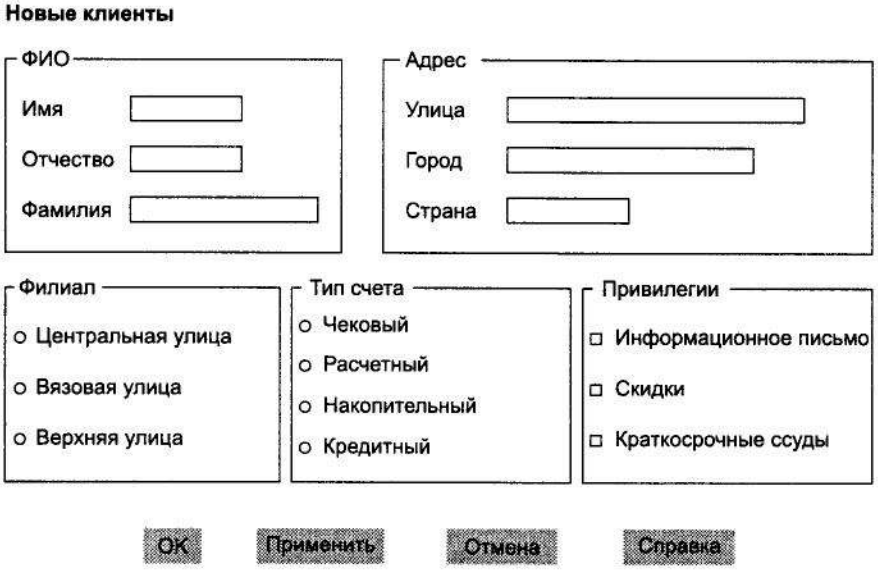


Рис. 3.13. Применение принципов хорошего экранного дизайна: улучшенный вид

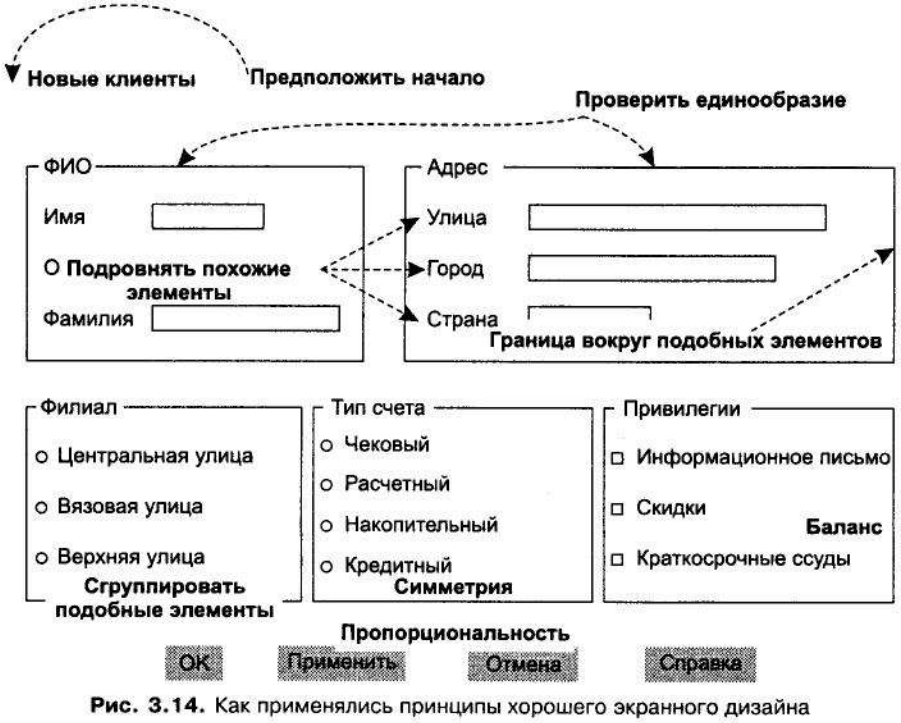


Рис. 3.14. Как применялись принципы хорошего экранного дизайна

Шаг 4 (выбор подходящего типа окна). Цели каждого пользовательского интерфейса могут обслуживаться наиболее эффективно одним или двумя конкретны-

ми типами окон. Пять основных целей графических пользовательских интерфейсов и типы окон, удовлетворяющие каждой из них, перечислены на рис. 3.15 и 3.16. Для типов окон использована терминология Windows, хотя они и являются типовыми.

1. *Цель:* показать свойства объекта
— Окно свойств

Свойства автомобиля 189	
Свойство	Значение
Марка	Toyota
Модель	Camry
Номер	893-8913-789014

2. *Цель:* получить дополнительную информацию для выполнения конкретного задания или команды
— Диалоговое окно

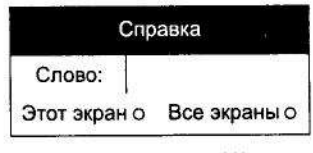
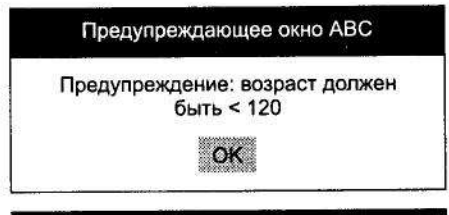
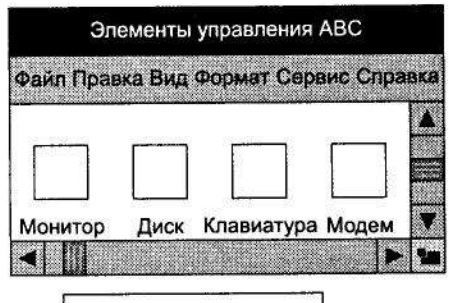


Рис. 3.15. Использование окон (1)

3. *Цель:*
предоставить информацию
— Окно сообщения



4. *Цель:*
предоставить набор элементов управления
— Окно с палитрой



5. *Цель:*
предоставить дополнительную информацию
— Всплывающее окно

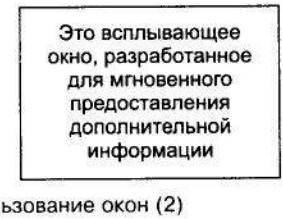


Рис. 3.16. Использование окон (2)

Шаг 5 (разработка системного меню). Ниже перечислены некоторые правила для создания главных меню, предложенные в [35]

- Сделать главное меню.
- Показать все уместные альтернативы (но только их).
- Привести структуру меню в соответствие со структурой задачи приложения.
- Минимизировать число уровней меню.

Пользователям нужен постоянный, понятный способ использования приложений, а отсюда и необходимость постоянного главного меню. Количество элементов в этом меню обычно должно быть от пяти до девяти, поскольку большинство из нас чувствует себя уютно с таким выбором. Например, текстовый редактор, в котором набиралась эта книга, имеет девять элементов меню: Файл, Правка, Вид, Вставка, Формат, Сервис, Таблица, Окно и Помощь. Элементов могло бы быть гораздо больше, поскольку места еще осталось довольно много: однако, нам, наверно, пришлось бы постоянно сканировать этот список в поисках необходимой нам команды, и эта лишняя нагрузка была бы результатом увеличенного списка. Элементы главного меню определяются задачей приложения — в нашем случае — редактирование текста. Поэтому, например, графические команды расположены во вторичном меню.

Шаг 6 (выбор подходящих устройств управления). Под устройствами управления здесь понимаются физические устройства, с помощью которых пользователи сообщают свои пожелания приложению. Сюда относятся джойстики, трекболы, графические планшеты, сенсорные экраны, мыши, микрофоны и клавиатуры.

Шаг 7 (выбор подходящих экранных элементов управления). Экранные элементы управления — это символы, появляющиеся на мониторе, с помощью которых пользователь передает программе вводимые данные и свои намерения. Сюда относятся значки, кнопки, текстовые окна, списки, изображенные на рис. 3.17. Правила организации экранных элементов управления в окне практически те же, что и для дизайна экрана в общем. Их число также обычно варьируется от пяти до девяти. Это число, однако, может быть увеличено в случае использования иерархии. Например, на рис. 3.13 имеется 20 возможностей для выбора, однако интерфейсом легко управлять, поскольку 20 элементов организованы в 6 групп.

Шаг 8 (организация и планирование окон). Правила для компоновки многочисленных окон аналогичны правилам для дизайна одиночных окон (в том числе симметрия, пропорциональность и т. д., см. шаг 3), но сюда включена также и организация окон — соприкасающиеся или каскадные. Последние термины проиллюстрированы на рис. 3.17.

Шаг 9 (выбор подходящих цветов). При использовании с умением и со вкусом цвет может обогатить экран. Использование цвета не делает автоматически пользовательский интерфейс более полезным или привлекательным, однако может легко испортить его. По выражению знаменитого дизайнера Поля Рэнда

«цвет — это воплощение сложности» [91]. Инженеры-программисты, не сотрудничающие с профессиональными дизайнерами, должны быть очень умеренными и консервативными в использовании цветов. Сначала попробуйте черно-белую схему. Если есть очевидная потребность в этом, добавьте один цвет. Убедитесь, что это помогает пользователю. Серьезно подумайте перед тем, как добавлять больше цветов. Наблюдение за грамотно сделанными программами, такими как широко используемые текстовые редакторы, может подсказать, как правильно использовать цвета. Можете быть уверены, что эти интерфейсы разрабатывали опытные профессионалы, и нетренированный программист может только выиграть от имитации подобных приложений. Синий цвет широко встречается в самых разнообразных окнах. Часто рекомендуется соблюдать симметрию цветов, и эта симметрия может быть нескольких видов. Например, текстовый редактор автора данной книги использует преимущественно три градации синего, встречающиеся в симметричном наборе стандартной цветовой палитры. Другие два используемых цвета — это желтый и, в меньшей мере, зеленый. Эти цвета используются в малых количествах, только акцентируя дополнительную функциональность, и не конкурируют с основными цветами, такими как черный и серый.

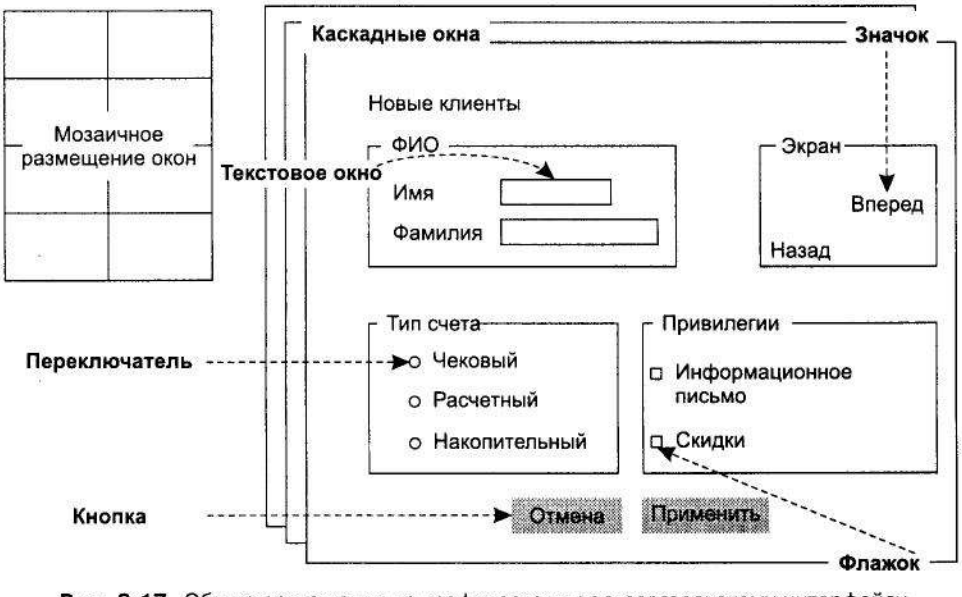


Рис. 3.17. Общие соглашения по графическому пользовательскому интерфейсу

3.3.5.2. Примеры черновиков требований

Иногда выполнение программы можно промоделировать посредством показа последовательности изображений графического пользовательского интерфейса. Например, можно дать представление об игре *Встреча*, показав последовательность снимков экрана. В любом случае, усовершенствование графического пользовательского интерфейса может обернуться продолжительным процессом общения с заказчиком.

Карандашный набросок графического пользовательского интерфейса для задания характеристик персонажа игры показан на рис. 3.18. Такие программные продукты, как Visual Basic и Java Beans, позволяют быстро создать графический пользовательский интерфейс. Желание сделать этот интерфейс частью финального продукта понятно, но это целесообразно, только если:

- ✦ язык графического пользовательского интерфейса является приемлемым языком для этой части приложения;
- ✦ инструмент построения графического пользовательского интерфейса генерирует поддерживаемый код.

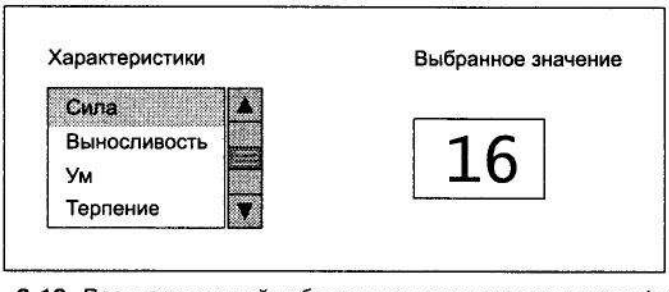


Рис. 3.18. Предварительный набросок пользовательского интерфейса для задания характеристик игрового персонажа

В противном случае графические пользовательские интерфейсы можно набросать просто с помощью инструментов рисования, как показано на снимке чернового экрана на рис. 3.19. Этот графический интерфейс подробно описан в главе 4.

После того как заказчику показали графический пользовательский интерфейс, он обычно начинает осознавать, что ему нужно нечто большее или что он хотел чего-то другого. В примере, показанном на рис. 3.18, заказчик мог бы решить, что графический интерфейс для изменения значений характеристик не подходит, так как общая сумма значений должна оставаться постоянной. Заказчик также, вероятно, заметит, что пользовательский интерфейс не очень привлекательный. Процесс утверждения графического пользовательского интерфейса очень интерактивен. D-требования предоставляют точную спецификацию графического пользовательского интерфейса, как показано в главе 4.

3.3.5.3. Другие интерфейсы

Помимо пользовательских интерфейсов, приложения часто должны взаимодействовать с другими системами; в этом случае интерфейс определяется в SRS. В качестве примера может послужить интернет-приложение, взаимодействующее с серверной резидентной CGI-программой (CGI — Common Gateway Interface, общий шлюзовой интерфейс). Формат, требуемый CGI-программой, должен быть установлен, например:

```
<URL>/cgi - bin/query?pg=<value>&dk=<value>
```

(*pg* и *dk* — параметры серверной функции из .../query)

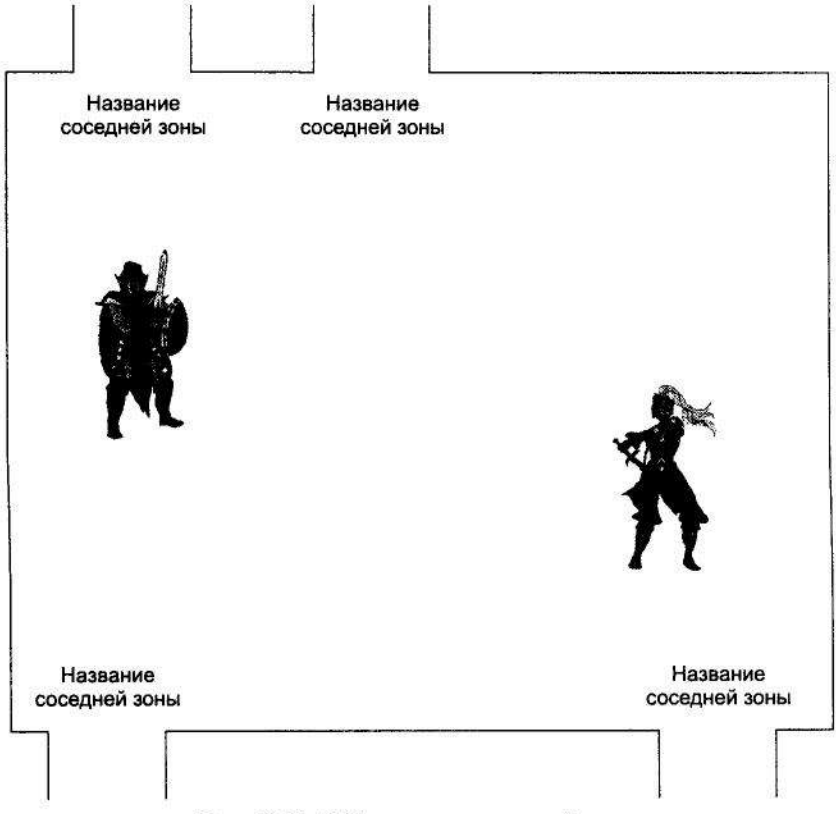


Рис. 3.19. Набросок экрана игры Встреча

Спецификации интерфейсов часто состоят из названий вызываемых функций, типа возвращаемого ими значения и типов аргументов, которые они требуют (их сигнатура): например, спецификация функции

```
float getValue(float principle, int numYears)
```

Интерфейсы могут также состоять из форматов сообщений или спецификации генерируемых и обрабатываемых событий.

3.3.6. Подведение итогов и руководство для формулирования С-требований

Подведем итоги способам формулирования требований заказчика. Ниже представлены четыре альтернативных формы для выражения требований заказчика. Выбор формы зависит от заказчика, а также от типа описываемого требования. Многие требования остаются сами по себе, например «пользователь должен иметь возможность устанавливать цвет вводимого текста: красный, синий или черный». Простой текст достаточен для формулирования таких требований. В главе 4 обсуждается их размещение в SRS.

ОДИН ИЗ СПОСОБОВ ВЫРАЗИТЬ ТРЕБОВАНИЯ ЗАКАЗЧИКА

- Если требование простое и стоит само по себе, выразите его четкими предложениями в соответствующем разделе SRS.
- Если требование представляет собой взаимодействие пользователя и приложения, выразите его с помощью варианта использования.
  1. Дайте имена вариантам использования.
  2. Определите *действующие лица*. Роль внешнего пользователя — обычно человек.
  3. Запишите последовательность действий пользователя и приложения.
  4. Минимизируйте ветвление.
  5. Используйте общую форму.
  6. Избегайте конкретных имен и значений. Например, вместо текста «Эд кладет $300» следует написать «клиент вводит размер депозита».
- Если требование затрагивает элементы обработки, каждый из которых получает и выдает данные, используйте диаграмму потоков данных.
  1. Определите элементы обработки (обычно высокого уровня), покажите их в кругах или прямоугольниках.
  2. Определите хранилища данных; покажите их как имена между двумя горизонтальными линиями.
  3. Покажите пути передачи данных между обрабатывающими элементами. Укажите типы данных, передаваемых в каждом случае.
- Если требование затрагивает состояния, в которых может находиться программа (или части программы), выполните следующие действия.
  1. Определите состояния (каждое состояние обычно определяют отглагольным существительным, например «Ожидание»), покажите их в прямоугольниках со скругленными углами.
  2. Укажите исходное состояние с помощью специальной стрелки.
  3. Определите события (происходящие вне рассматриваемой части системы), приводящие к переходу состояний, покажите их как помеченные стрелки.
  4. Определите вложенные состояния, покажите их как прямоугольники внутри прямоугольников.

Варианты использования широко применяются для описания требований заказчика, поскольку они показывают взаимодействие пользователя и программы. Если диаграмма переходов состояний показывает желания и нужды заказчика, и заказчик понимает эту диаграмму, использование диаграммы уместно. То же касается и диаграмм потоков данных. Техники потоков данных и переходов состояний широко используются при *проектировании*. При использовании их для спецификации требований присутствует некоторая опасность заняться проектированием вместо того, чтобы концентрироваться на требованиях. Например, если требуется, чтобы программа отслеживала потоки заказов внутри компании, уместной формой С-требований будет использование *диаграммы потоков данных* (DFD — Data Flow Diagram), показывающей процесс на высоком уровне, поскольку диаграмма потоков данных необходима для отображения того, что необходимо сделать. С другой стороны, давайте рассмотрим программу, которая должна возвращать результат выполнения сложной формулы. Диаграмма потоков данных, объясняющая процесс вычисления, была бы частью проектирования, а не требований.

3.4. Методологии и инструментальные средства для С-требований

Для выражения требований используются многочисленные методологии. Здесь приведен обзор некоторых из них.

Структурный анализ формализует потоки данных и функциональную декомпозицию. В частности, технология структурного анализа и проектирования (SADT — Structured Analysis and Design Technique) [92] являются систематизированным подходом к работе с системными спецификациями. SADT описывает проблему на самом высоком функциональном уровне как прямоугольник с входами, ограничениями и выходами («диаграмма контекста»). Это разлагается в следующий уровень диаграммы потоков данных, который затем аналогично разлагается дальше. В результате появляется иерархия диаграмм с возрастающим уровнем детализации потоков данных. Структурный анализ рассматривает приложение с точки зрения функциональности. Эта функциональность и реализована затем в иерархии функций.

Системы реального времени могут быть эффективно описаны с помощью диаграмм переходов состояний, используемых с объектной ориентацией или без (например, [108]). В частности, *Statemate* — это графический инструмент, использующий переходы состояний, опирающийся на работу Харела [41]. Он предоставляет основанный на состояниях метод, с помощью которого можно планировать и анализировать сложные системы. Например, современные суда имеют несколько резервных навигационных систем. В результате комбинирования системных аварийных состояний навигационные возможности на некоторых кораблях могут находиться в одном из огромного числа возможных состояний. Это может быть трудно понять без графического представления состояний (по личному опыту автора). Уже было рассмотрено в случае игры *Встреча*, что структуру игры можно легко понять с помощью системы переходов состояний, управляемой событиями. Большинство обозначений Statemate заимствованы в UML.

Система PSL/PSA (Problem Statement Language/Problem Statement Analyzer) (например, [106]) была одной из ранних систем для выражения требований. Исходные версии были основаны на текстовом представлении. Типичными заголовками компонентов были: ИМЯ ПРОЦЕССА, ОПИСАНИЕ, ГЕНЕРИРУЕТ, ПОЛУЧАЕТ, ЯВЛЯЕТСЯ ЧАСТЬЮ, ИСПОЛЬЗУЕТ. Описания компонентов хранятся в базе данных; их можно получить и обработать различными способами.

3.5. Быстрое прототипирование, исследование осуществимости и проверка концепции

Неудавшиеся программные проекты могут принести потери в миллионы долларов. В главе 2 мы обсуждали определение риска и его снижение как необходимые действия для предупреждения проблем с проектом. В этом разделе описан способ

сбора информации обратной связи по риску одновременно с процессом формулирования С-требований.

3.5.1. Быстрое прототипирование

Быстрое прототипирование — это частичная реализация целевой программы, в том числе обычно значительной части пользовательского интерфейса. Быстрое построение прототипа — полезный способ установить требования заказчика, а также определить и упразднить рискованные части проекта. Хорошее понимание может сэкономить дорогую работу и предупредить возможные будущие сложности.

Большие программы, такие как оборонные проекты на миллиарды долларов, используют дорогое прототипирование, так как их требования очень трудно удовлетворить. Что касается небольших проектов, здесь может быть достаточно простой графики, показывающей пользовательский интерфейс с помощью инструментов рисования, однако это зависит от природы проекта. Студенческие проекты часто остаются в выигрыше от прототипа, при условии что прототип довольно скромен. Он дает возможность команде опробовать свой процесс разработки до начала работы над самим проектом.

Чем детальнее прототип, тем легче понять требования заказчика. С другой стороны, прототипы сами по себе являются программами, поэтому чем детальнее прототип, тем он дороже. Первый взгляд на проблему решения, строить прототип или нет, показан на рис. 3.20. Таблица на рис. 3.20 показывает, например, что относительно недорогой прототип с большой ценностью должен быть создан. Под «большой ценностью» имеется в виду, что построение прототипа поможет заказчику лучше понять, продукт какого типа будет выпущен, помогает программистам лучше понять, какой продукт они должны выпустить.

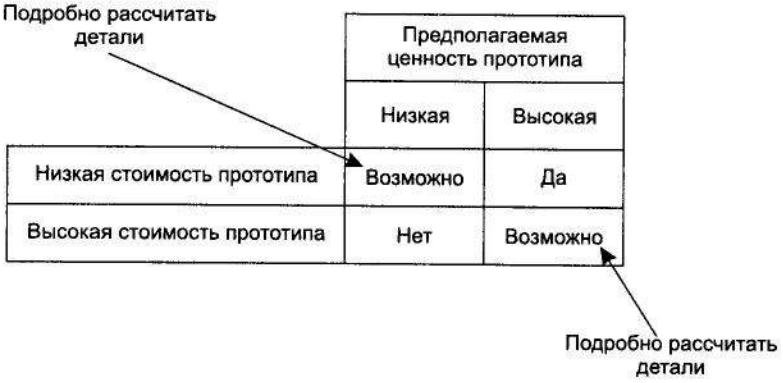


Рис. 3.20. Выгода от прототипа: грубая оценка

Многие случаи попадают в категорию «возможных» в таблице на рис. 3.20, и требуют использования метрик. Мы ищем оптимальный уровень затрат на прототип, как предполагается на рис. 3.21. По мере того как возрастают расходы на прототип, возрастает и его пригодность, но также возрастают и расходы из выделенного бюджета. В результате, вероятно, существует момент, в который за-

траты оптимальны (точка максимума на кривой), и некоторая точка, за которой деньги уже потрачены зря (где кривая пересекает горизонтальную ось).

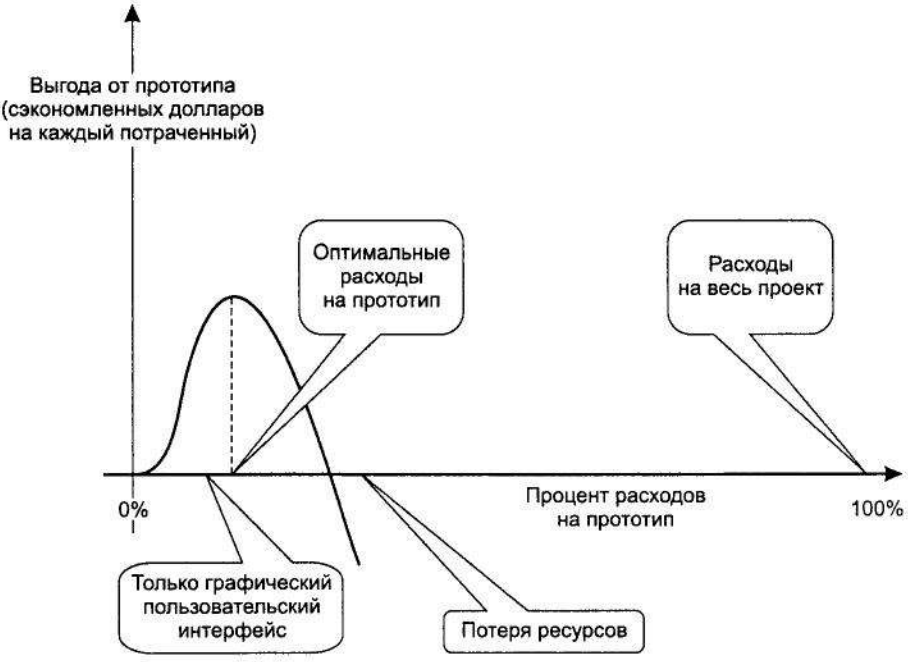


Рис. 3.21. Выгода от прототипа

Как пример представьте себе приложение для электронной коммерции, в котором компания-производитель одежды желает продавать товары через Интернет, хранить информацию о клиентах и предоставлять клиентам возможность получать свое фото в одежде из каталога. Финансовая оценка для разных уровней прототипирования для программы, продающей одежду, приведена в табл. 3.2. Для каждой из четырех характеристик, рассмотренных в прототипе, сделано несколько оценок: стоимость работы; процент работы, который будет повторно использоваться в самой программе (то есть не будет отброшен); и полная прибыль от прототипа. Под полной прибылью здесь понимается оценка того, что будет получено, если характеристика будет включена в прототип, но код не будет использован в программе. Например, мы подсчитали, что если прототип «примерка одежды» будет построен, это сэкономит минимум $20 000 при разработке. Оценка базируется на нижеследующих факторах.

- Предотвращение пустой траты времени на предложенные требования, которые, как видно из прототипа, не нужны (то есть минимум три ненужных требования из 100; на этап требований выделено $300 000, сэкономлено $9000).
- Разработка программного проекта «примерка одежды», что уменьшает риски разработки (то есть оценка того, что это сэкономит минимум одну человеко-неделю времени проектирования = $2000).

- Переработка, которая может возникнуть из-за изменения требований клиентом после того, как он увидит окончательный продукт (то есть переработка минимум трех требований по $3000 каждое = $9000).

Существует минимальная экономия, эквивалентная $9000 + $2000 + $9000 = = $20 000.

Таблица 3.2. Оценка программы по продаже одежды

Часть прототипа	Оценка стоимости	Общая прибыль без повторного использования кода		Процентная часть кода прототипа, повторно использованная в приложении
		Мин.	Макс.	
1. Экранные снимки пользовательского интерфейса	$10 000	$10 000	$80 000	50 %
2. Безопасность транзакций	$50 000	$10 000	$300 000	80 %
3. Завершение транзакций	$80 000	$10 000	$400 000	50 %
4. Примерка одежды клиентом	$120 000	$20 000	$140 000	30 %

Оценивая стоимость разработки прототипа, можно использовать техники оценки, аналогичные описанным в главе 2. Оценка повторного использования кода может быть выполнена путем определения классов прототипа и решения того, какие из них будут использованы в самой программе.

Такая оценка состоит из исследования стоимости небольших частей, что все же является сложной задачей. Определение минимума и максимума такой оценки может несколько упростить этот трудный процесс.

Как только оценки сделаны, выполняется несложная часть вычисления лучшего и худшего сценария. Это показано в табл. 3.3. Минимальное значение выгоды получается из самой пессимистичной комбинации — высоких затрат, низкой общей прибыли, и низкого процента повторного использования. Максимальная выгода рассчитывается аналогично. Например, максимальная выгода (для наиболее оптимистичной альтернативы) для части прототипа «снимки пользовательского интерфейса» будет:

[максимальная оцененная прибыль] – [минимальная оцененная стоимость] = = $80 000 – [(минимальная оцененная стоимость) × (процент неиспользования)] = = $80 000 – [$10 000 × 50 %] = $75 000.

Усреднение — это один из способов работы с разницей между худшими и лучшими случаями. В результате получаем положительную выгоду для всех предложенных частей прототипа за исключением части «примерка одежды», которая оценена в –$4000: общий убыток $4000. Это приведет в дальнейшем к относительно низкой прибыли, высокой стоимости разработки и низкому вторичному использованию.

Таблица 3.3. Расчет выгоды от прототипа для программы по продаже одежды

Часть прототипа	Оценка стоимости	Общая прибыль без повторного использования кода		Процентная часть кода прототипа, повторно использованная в приложении	Чистая выгода		
		Мин.	Макс.		Мин.	Макс.	В среднем
	B	D	E	C	D–(1–C)B	E–(1–C)B	
1. Экранные снимки пользовательского интерфейса	$10 000	$10 000	$80 000	50 %	$5000	$75 000	$40 000
2. Безопасность транзакций	$50 000	$10 000	$300 000	80 %	$0	$290 000	$145 000
3. Завершение транзакций	$80 000	$10 000	$400 000	50 %	–$30 000	$200 000	$85 000
4. Примерка одежды клиентом	$120 000	$20 000	$140 000	30 %	–$64 000	$56 000	–$4000

Здесь можно посоветовать развивать сам прототип до получения программы, но это должно быть запланировано, а не получаться спонтанно. По своей природе прототипы быстро строятся и редко документируются. Они часто разрабатываются с помощью языков, дающих быстрый результат, но которые могут не подходить для самой программы. Довольно часто разработчики показывают клиенту прототип, написанный на таком языке как Visual Basic или PERL. Часто такой прототип производит на клиента сильное впечатление, и клиент ошибочно считает, что работа близка к завершению. Неудачным следствием этого может быть решение разработчика довести прототип до финальной программы. За исключением случая, когда прототип и разрабатывался с такой целью, проект может потребовать гораздо больше времени, если такая попытка будет все-таки предпринята. В качестве аналогии можно привести пример, когда в качестве спецификации дома клиенту показывают прототип дома, построенный из соломы, но создающий хорошее впечатление о виде будущего дома клиента издалека. Однако, согласитесь, немногие захотят строить сам дом на основе соломенной конструкции.

Быстрая разработка приложений (RAD — Rapid Application Development) основана на прототипировании. По мнению автора, название «Быстрая разработка приложений» вводит в заблуждение. Если бы RAD действительно была способом *быстрой* разработки *настоящих* приложений, то все бы так и работали. Этот волшебный способ стал бы нормой и не мог бы рассматриваться как «быстрый».

3.5.2. Исследование осуществимости

Иногда неочевидно, могут ли вообще быть реализованы предложенные требования. Другими словами, существует риск для всего проекта, а не риск, касающийся конкретных требований. Вдобавок, проект не будет осуществим, не будет стоить проделанной работы, если риск реально существует. В таких случаях, возможно, потребуется провести исследования осуществимости требований. Эти исследования представляют из себя частичную разработку или моделирование программы. Например, представьте себе осуществимость интернет-программы *Встреча* на Java. Допустим, мы предполагаем, что программа будет настолько медленно работать, что игра не будет интересной. Исследование осуществимости может состоять в передаче сообщений с нужной частотой от некоторого числа игроков, но с неосмысленным содержанием. По этим измерениям можно будет оценить задержку.

Численные модели стоят дорого, поскольку они сами по себе являются программами, требующими от программистов выполнения всех артефактов, описанных в этой книге, таких как, например, создание отдельного SRS! Автор однажды соприкоснулся с моделированием большой системы, находящейся в разработке. Никто не воспринимал требования к модели серьезно, поскольку это «не был реальный продукт». В результате стоимость поддержки и использования модели выросла до астрономических размеров. Например, чтобы выполнить изменения, приходилось сначала искать сотрудника, «знакомого с системой». Оценки осуществимости часто встречаются в крупных оборонных программах, требующих дорогостоящего программного обеспечения и оборудования.

Когда мы просто не можем решить, имеет ли смысл заняться предлагаемой программой, иногда можно построить *доказательство концепции*. Это частичная реализация приложения или программа, похожая на предлагаемую к разработке.

Например, перед тем как флот США создал поколение Aegis навигационных систем, была создана полномасштабная корабельная система с соответствующим программным обеспечением и оборудованием. Это доказательство концепции имеет целью убедить флот в большой вероятности того, что концепции, предполагаемые для Aegis, могут быть реализованы за приемлемое время и деньги.

3.6. Корректировка проекта для отображения анализа С-требований

Набор документов — это «живая» сущность: его нужно регулярно обновлять во время работы над проектом. В типичном случае завершение одной из фаз затрагивает несколько документов, которые нужно обновить. Управление этим процессом, часто лежащее вне возможностей организации, является ключом к нормальному ходу проекта.

3.6.1. С-требования и размеры проекта

Для больших проектов анализ требований клиента достаточно формален и организован. Например, министерство обороны США часто публикует запрос о предложениях только для того, чтобы разработать SRS. Такой запрос содержит только высокоуровневое описание проекта. Этот запрос можно представить себе как С-требование, а продукт фирмы-исполнителя — как D-требование. Нескольким сотрудникам министерства обороны дается задание работать с фирмой-исполнителем над SRS, причем эта фирма не обязательно должна быть фирмой, выбранной для проектирования и реализации самой программы. Чтобы убедиться в выполнимости требований, проводятся многочисленные собрания. На эти собрания приглашаются сотрудники фирмы-исполнителя, государственные чиновники-специалисты и менеджеры, а также офицеры морского флота или воздушных сил США, и т. д. SRS может разрастись до тысяч страниц.

3.6.2. Влияние анализа С-требований на план проекта

Как только С-требования собраны, SPMP можно обновить (табл. 3.4). Такое обновление происходит в течение всего жизненного цикла программы.

Таблица 3.4. Обновление плана проекта после получения С-требований

	Статус после создания исходного черновика	Результаты обновления SPMP после получения С-требований
Вехи	Исходный	Больше вех, более конкретно
Риски	Определить исходный риск	Уменьшить определенные ранее риски, определить новые риски теперь, когда о проекте больше известно

продолжение

Таблица 3.4 (*продолжение*)

	Статус после создания исходного черновика	Результаты обновления SPMP после получения C-требований
План	Очень приближенно	Предварительный план проекта
Кадры	Назначить разработчиков C-требований	Назначенные для анализа D-требований разработчики
Оценка стоимости	Очень приближенно	Первые оценки на базе содержания работы

Результирующее расписание часто бывает похожим на содержание рис. 3.22, с большим количеством деталей на момент чернового варианта (глава 2), но все еще не очень подробным.

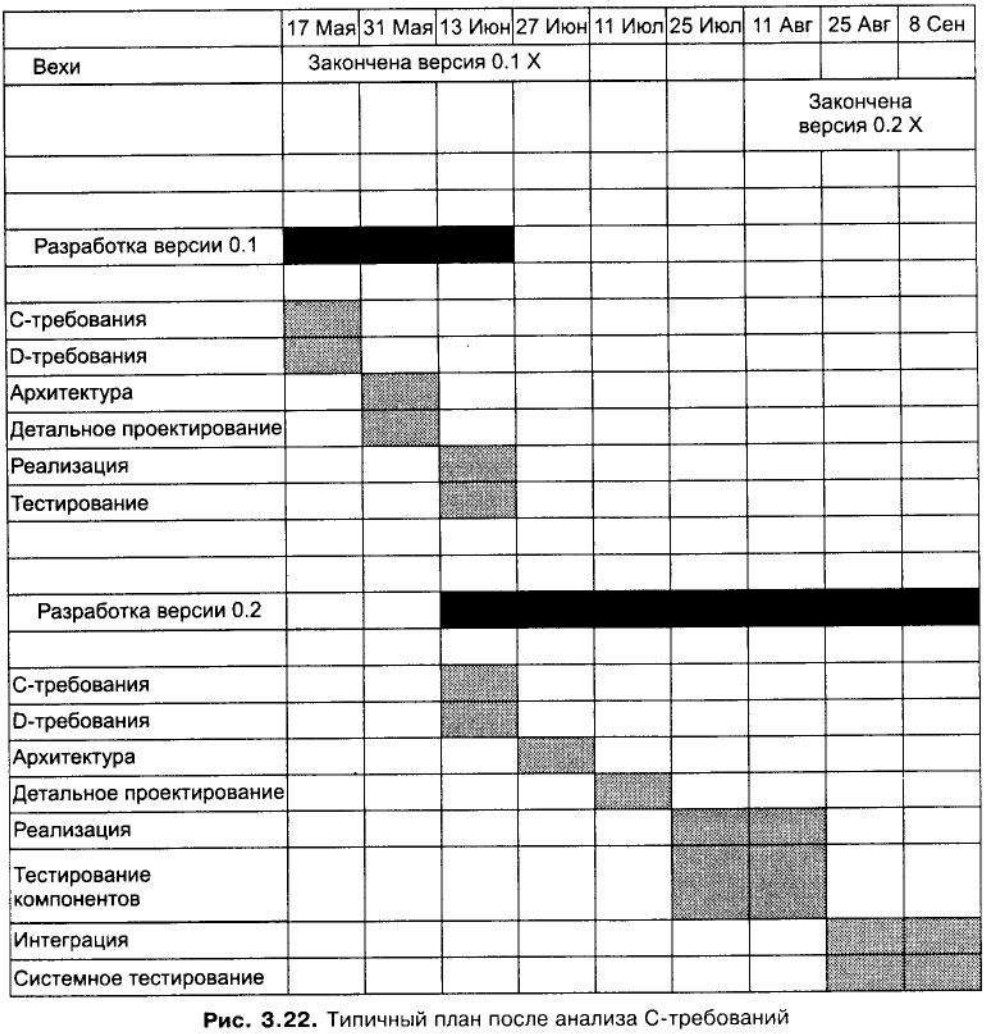


Рис. 3.22. Типичный план после анализа C-требований

Хотя процесс анализа требований может иметь много итераций в течение жизни проекта, у такого итеративного процесса есть свои практические ограничения. Клиенту часто нужно знать стоимость работы заранее, однако разработчики обычно договариваются о стоимости только после замораживания требований. Это ограничивает степень, до которой происходят итерации в требованиях.

Оценка стоимости может быть улучшена после анализа С-требований. Основное улучшение происходит от улучшенного понимания разработчиками природы и области применения программы. Функциональная оценка может быть более завершенной, и то же касается оценок планов и трудозатрат. Прямые восходящие оценки могут быть также улучшены.

SRS и ее разделы являются *элементами конфигурации программы*, как определено в разделе 1.7.3.1. Типичный моментальный снимок статуса конфигурации такой спецификации показан в табл. 3.5. Обратите внимание, что номер редакции, указанный для заголовка, не может быть меньше, чем номера его подчиненных элементов конфигурации. Например, номер редакции *Введения* должен быть как минимум 3, поскольку подраздел *область применения* прошел три редакции, и каждая из них должна была привести к отдельной редакции *Введения*.

Таблица 3.5. Типичное состояние статусов конфигурации SRS

SRS: выпуск 2.1 от 27.05.98			**Номер редакции**
1. Введение			3
	1.1. Цель		2
	1.2. Область применения		3
	1.3. Определения, термины и сокращения		2
	1.4. Ссылки		1
	1.5. Обзор		3
2. Общее описание			4
	2.1. Перспективы продукта		4
		2.1.1. Системные интерфейсы	2
		2.1.2. Пользовательские интерфейсы	1
		2.1.3. Аппаратные интерфейсы	1
		2.1.4. Программные интерфейсы	4
		2.1.5. Коммуникационные интерфейсы	1
		2.1.6. Ограничения по памяти	4
		2.1.7. Операции	1
		2.1.8. Требования по адаптации	4

продолжение

Таблица 3.5 (*продолжение*)

SRS: выпуск 2.1 от 27.05.98		Номер редакции
	2.2.Функции продукта	3
	2.3. Пользовательские характеристики	3
	2.4. Ограничения	3
	2.5. Предположения и зависимости	4
	2.6. Распределение требований	1
3. Конкретные требования (глава 4)		6
4. Сопровождающая информация		3

3.7. Будущие направления и подведение итогов по С-требованиям

3.7.1. Будущие направления

Поскольку анализ требований критичен для кампаний, разрабатывающих программное обеспечение, в течение многих лет проводились исследовательские работы в этом направлении. Одно из направлений исследований, «исполняемые спецификации», затрагивает определение требований так, чтобы их можно было автоматически транслировать в исполняемый код. Поскольку стиль требований является декларативным (утверждения о фактах), исполняемым спецификациям требуется процесс, который преобразовывал бы их декларативные утверждения в командные сообщения для компьютеров. Компьютерный язык Пролог является декларативным и исполняемым, поэтому он и использовался для многих приложений. Он доказывает, что набор утверждений, похожих на требования («что» вместо «как»), может быть исполняемым.

В течение нескольких лет считалось, что инструменты автоматизированной разработки программ (CASE) могут значительно упростить процесс отслеживания требований. Использование таких инструментов прошло через стадии оптимизма и пессимизма. Другое направление исследований (например, [102]) идет гораздо дальше использования идей вариантов использования, предоставляя библиотеку с высокоуровневыми объектно-ориентированными образцами проектирования, с которыми инженер по требованиям пытается сопоставить проект. Некоторые считают, что многочисленные требования могут быть составлены из библиотек уже существующих вариантов использования. Было несколько попыток применить искусственный интеллект к анализу требований, в основном в форме интеллектуальных агентов; однако ни один из них не используется широко и по-

всеместно. Растущее взаимодействие пользователей с компьютерами произвело эффект на анализ требований, открыв новый спектр вопросов (например, [109]).

3.7.2. Подведение итогов по С-требованиям

В этой главе описан процесс, посредством которого требования клиента к продукту добываются и записываются способом, понятным клиенту. Требования клиента документируются в форме разделов 1 и 2 SRS по стандарту IEEE 830-1993. Для этого используются различные техники выявления и формулирования С-требований.

Анализ требований представляет собой перспективный процесс, поскольку понимание того, что нужно сделать, требует значительного активного извлечения информации у клиента. Идеи этой главы резюмированы ниже.

- С-требования предназначены преимущественно для клиента.
  - Включая пользовательский интерфейс.
- D-требования — для разработчиков.
- Необходимо использовать стандартные SRS (например, IEEE).
- Варианты использования очень эффективны.
  - Повторное использование в тестовых примерах.
- Диаграммы переходов состояний и потоков данных могут также быть эффективными спецификациями.

Руководство по учебному проекту. С-требования для видеоигры Встреча

В этом разделе объясняется, как принципы, описанные в главе, используются на практике на примере игры *Встреча*. Автор рекомендует обратить внимание на врезки «Один из способов...» в этой главе, которые являются общим руководством к работе.

Этап 1. Подготовка

Халл Фурнесс, избранный ответственным за требования, должен был организовать анализ требований. Что касается организации проекта, Халла заменила Карен Петерс. Они решили собрать требования за два этапа. На первом — преимущественно с точки зрения клиента (С-требования), а на втором — преимущественно с точки зрения разработчиков (D-требования).

[В этом руководстве описывается процесс для С-требований. Учебный проект в следующей главе описывает D-требования.]

Халл и Карен подготовились к сбору информации. Они классифицировали процесс по следующим этапам: *подготовка, интервью, письменное изложение*

и *проверка*. Метрики, которые они выбрали, в основном были продиктованы политикой фирмы.

- Затраченное время.
- Число страниц сформулированных С-требований.
- Самооценка артефактов по шкале 1–10 (не установлена политикой фирмы).
- Дефекты, обнаруженные во время инспекции.

Читателю предлагается обратить внимание на раздел 4 данного руководства, в котором эти метрики организованы в таблице.

Карен убедилась, что система для записи и отслеживания дефектов установлена, и что Халл снабжен документацией относительно ее использования.

Инвесторы компании посчитали видеоигры многообещающим предприятием и решили предоставить начальные инвестиции на анализ требований и прототип. Теперь дело за Халлом и Карен — они должны определиться, с кем следует разговаривать относительно С-требований. Халл понял, что члены команды плохо знакомы с видеоиграми. Он решил взять интервью у людей, которые часто в них играют, и которым будет интересно потратить свое время за скромное вознаграждение. Он связался с Бетти Симс, президентом Международной ассоциации любителей игр, увлеченным игроком, которая видела яркое будущее для видеоигр как средств развлечения и образования. Бетти также была знакома со многими любителями игр. Халл и Карен решили написать спецификацию требований на базе информации, полученной от Бетти, а затем показать эту спецификацию остальным. В это время остальная часть команды должна была разведать другую информацию.

На еженедельном собрании Халл представил нижеследующий план анализа требований.

Неделя 1:

Халл и Карен: интервью с Бетти; начать писать черновик С-требований.

Ферн и Эл: поиск других источников информации по требованиям.

Неделя 2:

Ферн и Эл: доклад о найденных кандидатах на еженедельном собрании.

Команда: выбрать одного-двух человек из списка Ферна и Эла для формулировки требований.

Халл и Карен: завершить черновик С-требований, связаться по электронной почте с Бетти для ее комментариев; организовать интервью с выбранными дополнительными людьми; послать им существующую спецификацию; создать законченный документ требований для 1 итерации; передать его под управление конфигурациями.

Неделя 3:

Команда: утвердить SRS для первой итерации.

Халл и Карен: провести интервью с запланированными людьми; отредактировать и расширить спецификацию; связаться по электронной почте

со всеми интервьюерами; упорядочить их ответы; отредактировать документ, оставить некоторые вопросы для рассмотрения в команде; запланировать анализ D-требований [следующая глава].

Неделя 4:

Команда: предоставить информацию для черновика SRS; утвердить план для анализа D-требований [следующая глава].

Халл и Карен: написать SRS и послать ее по электронной почте всем интервьюерам.

Неделя 5:

Халл и Карен: решить вопросы, затронутые интервьюерами; записать результаты; разослать по электронной почте команде; начать процесс разработки D-требований [следующая глава].

Команда: проверить С-требования.

Несмотря на издержки, Халлу кажется важным, чтобы вся команда проверила С-требования из-за важности документа. Обычно планируется проверка документа группой в составе трех человек.

Халл спланировал первое интервью с Бетти в комнате 1428 с 10:00 до 11:30. Он послал ей короткое письмо, описывающее историю проекта и создал следующий очень простой план проведения встречи:

10:00–10:15. Халл: мотивы проекта.

10:15–11:30. Интервью с Бетти: требования заказчика.

Халл решил не предоставлять больше подробностей, поскольку он хотел, чтобы требования Бетти повлияли на оставшуюся часть собрания.

Этап 2. Интервью с заказчиком

Халл и Карен приехали на интервью с Бетти, имея хорошее звукозаписывающее оборудование. Бетти не могла понять, почему кто-то может хотеть создать видеоигру, если она не будет соперничать с лучшими играми в этом жанре. Халл объяснил, что это будет лишь первым шагом, предоставляющим команде возможность получить опыт в этой области программирования. Команда будет себе представлять требуемый спектр работ, а также сможет продемонстрировать инвесторам, что может быть достигнуто в рамках заданного объема финансирования. Среди других мотивов — определить, оправданы ли идеи о том, что видеоигры имеют потенциально широкий спрос и применимы в образовательной системе. После этого собрание больше сфокусировалось на рассмотрении деталей. Магнитофон был включен, и Халл с Карен начали подробно записывать.

Бетти считала, что ролевые игры (а не «стрелялки») являются многообещающими, поскольку они расширяют сообщество игроков. Она говорила о минимальных возможностях, которые должен иметь прототип. Сюда относятся области, в которых игровые персонажи будут вступать в контакт, способы попасть из одной зоны в другую, способы создания взаимодействия между персонажами и события, происходящие во время контакта. Халл и Карен пытались отделять

проблемы от характерных свойств игры по мере их возникновения, разнося их по столбцам «критично для первой итерации», «можно отложить» и «другое» (то есть использовали метод отбраковки). Важность требований, перечисленных в графе «другое», будет определена позднее.

Получив сценарий требований, описанных Бетти, Халл сфокусировался на получении примеров от нее. Он попросил Бетти описать типичные сценарии для игры. Бетти описала, что происходит при взаимодействии двух персонажей. Карен делала заметки и выделила эту часть в частный пример — последовательность действий, предпринимаемых игроком и (или) игрой, и затем прочитала то, что получилось, Бетти.

Бетти не могла придумать никакие другие сценарии. Халлу показалось, что их может быть больше, и он спросил, как должна начинаться игра. В результате сформировался еще один вариант использования. Третий вариант, который они записали, объяснял, как игрок перемещает свой персонаж из одной зоны в другую. Этих трех примеров, по мнению собеседников, будет достаточно для начала. Халл и Карен посчитали, что могут быть еще и другие примеры, но они придумают их позже.

Бетти, Карен и Халл вместе набросали несколько экранных изображений. На одном была изображена типичная встреча, на другом — экран для ввода характеристик игровых персонажей. Состоялась дискуссия относительно ракурса игрока, то есть того, как игрок видит игровую сцену. Бетти хотела иметь ракурс «из глаз». Карен считала, что требуемая сложность для такого ракурса выведет проект за рамки бюджета. Пришли к соглашению на модифицированном виде сверху, подходящем для прототипа. Это было отображено на черновиках изображений экранов. Однако все согласились, что потребуется значительная доработка пользовательского интерфейса.

В результате создания черновых вариантов интерфейсов Карен показалось, что игру можно понять только посредством описания состояний. Бетти не была знакома с этим термином, но она довольно уверенно использовала термин «режимы» типичной ролевой игры, что оказалось той же идеей. Карен и Халл набросали требуемые режимы игры и просмотрели вместе с Карен, каким образом игра переходит из одного состояния в другое.

Халл кратко рассмотрел дальнейшее развитие игры, проанализировав потоки данных, но скоро пришел к выводу, что рассмотрение потоков данных немного добавило к уже имеющейся информации.

Карен просмотрела свои заметки вместе с другими. Несколько моментов стоило исправить, но относительно общего описания игры участники дебатов пришли к соглашению.

Этап 3. Написание спецификации требований к программному обеспечению (SRS)

Халл и Карен разбили задачу написания SRS на части. Они использовали стандарт IEEE SRS, разделы 1 и 2 (раздел 3 состоит из детальных требований, процесс для которых обсуждается в руководстве к учебному проекту в главе 4). Чтобы

избежать конфликтующих записей, они постарались сделать их разделы как можно более независимыми. Халл вспомнил свой предыдущий проект, где команда потратила очень много времени на согласование частей, написанных разными людьми, в то время как было бы гораздо быстрее, если бы всем этим занимался один человек.

Они обсудили, как лучше расставить приоритеты по требованиям, поскольку становилось очевидно, что в противном случае список требований станет гораздо длиннее того, что команда сможет реально проработать. Халл хотел упорядочить все требования, но Карен указала, что затраченные на это силы будут неоправданны: большинство важнейших требований будет и так выполнено, так что их точный порядок не так важен. Почти ничто из малоприоритетных выполнено не будет, поэтому время, потраченное на них, также пропадет. Они решили использовать метод отбраковки, чтобы упорядочить требования по трем категориям: *необходимые*, *необязательные* и *желательные* (что попросту означает «ни первое, ни второе»). Они также решили, что может потребоваться позднее упорядочить необходимые требования. Это сэкономило им много времени. Они описали свою схему классификации в разделе 2.6 SRS («Распределение требований»).

На раздел 2.1.1 (концепция операций, содержащая диаграмму переходов состояний игры) у Халла ушла большая часть времени, так как ему пришлось перевести неформальные комментарии Бетти в четкую форму. Они попытались использовать перекрестные ссылки для некоторых частей SRS, с соответствующими тестами, даже хотя тесты все еще были на стадии набросков. Это помогло разъяснить требования самим себе. Когда Бетти посмотрела на тест к разделу 2.1.1, она поняла, что Карен и Халл не поняли некоторые ее высказывания. В частности, когда игра находится в состоянии *Оповещение* и в зону персонажа игрока входит внешний персонаж, тест не предусматривал никаких событий. Бетти посчитала это неправильным и назвала это возможностью для игрока заблокировать игру. Найденный дефект был добавлен в соответствующий список дефектов с категорией *Важно*.

Карен подготовила черновик пользовательского интерфейса. Для этого она воспользовалась инструментом PowerPoint, вместо того чтобы создавать его с помощью Java — целевого языка. Она посчитала PowerPoint подходящим, поскольку в этой части SRS пользовательский интерфейс предполагался черновым — подробно пользовательский интерфейс определен в разделе 3 — и в любом случае он, скорее всего, должен был сильно измениться. Это помогло Халлу и Карен показать черновики Бетти и остальным, получить обратные отзывы и затем точно определить пользовательский интерфейс для D-требований.

Этап 4. Завершение

Разделы 1 и 2 SRS были посланы Бетти по электронной почте. Она обнаружила, что Халл и Карен включили только два из трех вариантов использования, а третий вариант, описывающий движение персонажа игрока, был пропущен. Этот дефект был записан с высоким приоритетом.

Бетти была удивлена, увидев, что в спецификации не были отражены некоторые вопросы, важность которых, как ей казалось, она четко объяснила. Она поспешила проверить, отражены ли в спецификации требования, которые она вскользь упомя-

нула, но затем поняла, что это уже не так важно. К последнему относилась возможность игрока изменять внешний вид персонажа, не останавливая ход контакта. У нее были многочисленные комментарии, на большинство из которых Халл и Карен ответили и некоторые из которых были добавлены в список дефектов.

Халл разослал членам команды разделы 1 и 2 SRS, чтобы они подготовились к проверке.

Лидер команды Эд слышал об Арлане Ховарде, специалисте по маркетингу, хорошо знакомом с индустрией видеоигр. Финансовые поручители пожелали профинансировать дальнейший анализ требований на уровне заказчика, и Халл с Карен приготовились к встрече с Ховардом. Ховард не мог уделить им более получаса времени, поскольку был очень занят. Карен разработала список вопросов и тем с приоритетами и послала его вместе с существующим черновиком глав 1 и 2 SRS. Они планировали завершить написание С-требований после разговора с Ховардом.

Команда также планировала процесс разработки D-требований [руководство учебным проектом в главе 4].

Этап 5. Метрики и итоги

С-требования были проверены всей командой, после чего были записаны все дефекты.

На следующем недельном собрании Халл и Карен подвели итог метрике (табл. 3.6, звездочкой помечены организационные нормы этого проекта). Команда согласилась с приведенными наблюдениями.

Таблица 3.6. Подведение итогов

	Подготовка	Интервью	Записи (результаты исследования)	Проверка	Всего
Затраченное время (минуты)	200	170	270	250	14,8 часа
Для затраченного времени	200/890 = 22 % // 20 %*	170/890 = 19 % // 23 %*	270/890 = 30 % // 27 %*	250/890 = 28 % // 29 %*	
Результирующее количество			15 страниц		
Производительность (время/количество)			15/14,8 = 1,01 страниц в час // 0,95*		
Самооценка качества (1–10)	9	5	9	2	
Процент дефектов			1,3 на страницу // 1,01 на страницу*		

	Подготовка	Интервью	Записи (результаты исследования)	Проверка	Всего
Улучшение процесса	Потратить на 20 % меньше времени на подготовку	Более равномерно распределить интервью между разными людьми	Исходный материал хорошо написан, но его следует более подробно проверить перед исследованием	Потратить ±30 % больше времени на пересмотр	

Упражнения

Ответы и подсказки для упражнений, помеченных символами «о» или «п», приводятся в конце этой главы.

Вопросы для проверки

П3.1°. В чем разница между C-требованием и D-требованием?

П3.2°. В чем достоинства и недостатки разделения требований на категории C и D?

П3.3°. Что такое вариант использования?

П3.4°. Является ли следующий текст вариантом использования: «Система будет предоставлять совет начинающему пользователю Windows относительно того, как следует выполнять операции Windows».

П3.5°. Для какой из следующих программ имеет смысл создавать законченный прототип (да / нет / возможно):

1) система отслеживания посылок в большой почтовой компании;
2) простая система хранения информации о частных коллекциях CD-дисков;
3) система поддержки пенсионных счетов для маленькой фирмы.

П3.6°. Какое из следующих приложений требует исследования выполнимости:

1) база данных для хранения и просмотра информации о сотрудниках;
2) система, автоматически преобразующая абзац в одно предложение.

Общие упражнения

О3.1ⁿ. Брэкетт считает, что чем больше ограничений наложено на программу, тем меньше можно полагаться на людей как на источник требований (см. его график на рис. 3.4, сравнивающий «приближенный процент требований, собранных у людей», с «типом программы»). Можете ли вы придумать программу, не соответствующую этому графику?

О3.2. Предположим, что вы пытаетесь описать автомобильную программу, сообщающую статус системы стартера на приборную панель. Как бы вы описали ее в общих терминах?

О3.3. На не более чем трех страницах сформулируйте требования к приложению, отслеживающему штрих-коды счетов-фактур компании.

О3.4. Вашему заказчику нужно определить пользовательский интерфейс. Обсудите достоинства и недостатки нижеследующих средств для выполнения этой задачи в контексте программы (большой или маленькой) и природы интерфейса (сложной или простой).

1. Сделайте черновик, используя рисунки от руки — выполняются вами или художником.
2. Сделайте черновик, используя графические инструменты, такие как Paint или PowerPoint.
3. Используйте особенности целевого языка программирования для построения графического пользовательского интерфейса.

Упражнения в команде

К3.1. Напишите С-требования для программы по выбору разработчиков. Следуйте форме IEEE 830-1993. Проследите, сколько времени у вас займет выполнение этого упражнения. Решите, какая часть требований была осознана командой. Подсчитайте, сколько времени займет получение 95% требований. Сформулируйте, как можно было бы улучшить процесс, использованный вами. Будьте конкретны и предоставьте примеры.

К3.2.

1. Определите человека вне команды, которому необходима небольшая программа. У этого человека вы будете узнавать С-требования, а затем покажете их ему.
2. Вместе с вашим «заказчиком» определите метрику того, как он будет оценивать ваши С-требования. Также определите временные рамки для интервью (например, 1/2 часа).
3. Проведите интервью с заказчиком и напишите С-требования.
4. Попросите заказчика оценить и прокомментировать ваши С-требования в соответствии с выбранной метрикой.

Подсказки

П3.1. Найдите сильно ограниченную программу, требования которой следует собрать преимущественно у людей.

Ответы

П3.1. С-требования выражают требования в форме, удобной заказчику, и состоят преимущественно из высокоуровневого описания. Форма D-требований удобнее для разработчиков. D-требования представляют собой более подробную форму С-требований.

П3.2. Заказчики и разработчики имеют разные потребности в требованиях. К преимуществам разделения C- и D-требований относится тот факт, что они быстрее удовлетворят эти разные потребности. К недостаткам относится возможность того, что две формы описания не будут совпадать.

П3.3. Вариант использования — это последовательность взаимодействий пользователя и программы в типичной ситуации.

П3.4. Это не вариант использования, поскольку здесь не продемонстрирована последовательность действий, выполненных приложением и пользователем приложения.

П3.5.

1. Отслеживание посылок: да.
2. Собрания CD-дисков: вероятно, слишком малое приложение, чтобы оправдать прототип.
3. Пенсионные счета: возможно (нет, если программа предполагается для одного заказчика; возможно, если предполагается подавать ее как продукт многим заказчикам).

П3.6.

1. Система баз данных: вероятно, слишком стандартно, чтобы проводить исследования выполнимости.
2. Автоматическое резюме: да.

Пример. Спецификация требований к программному обеспечению (SRS) для видеоигры Встреча, часть 1

[Примечание для студентов. Использование стандарта для написания SRS поможет охватить все аспекты требований, о которых следует знать читателю, а также предоставит общепризнанную структуру. Существует несколько стандартов, однако мы сконцентрируемся на стандарте IEEE. Полный стандарт IEEE 830-1993 можно найти в [56]. Большинство организаций позволяют изменять стандарт, чтобы подогнать его к конкретному использованию. Использованный ниже шаблон изменяет стандарт, пропуская некоторые менее важные разделы и добавляя разделы с идеями операций и вариантов использования. Студент может сравнить заголовки нашего примера со стандартом, показанным в разделе 3.1.4.]

[Примечание для студентов. Часть главы, в которой рассматривается пример, а именно разделы 1 и 2, описывает требования заказчика (C-требования). Остальная часть документа — разделы 3 и 4, содержащие конкретные D-требования, — предоставлена в примере в конце главы 4. Не забудьте, что C-требования не имеют целью быть подробными в достаточной степени для проектирования и реализации: для этого существуют D-требования.]

История версий этого документа:

- x/yy/zzz Халл Фурнесс: исходный черновик.
- x/yy/zzz Карен Петерс: проверка технической точности; изменения текста.
- x/yy/zzz Халл Фурнесс: проверка всего документа, небольшие улучшения.
- x/yy/zzz Карен Петерс: проверка документа, внесение предложений.
- x/yy/zzz Карен Петерс: перемещение вариантов использования в раздел 2.2.
- x/yy/zzz Халл Фурнесс: коррекция стиля, смысл не изменен.

1. Введение

1.1. Цель

[Примечание для студентов. Цель документа в целом (а не цель программы).]

Этот документ предоставляет все требования для видеоигры *Встреча*. Части 1 и 2 предназначены преимущественно для заказчиков приложения, но также будут интересны инженерам-разработчикам, разрабатывающим или поддерживающим его. Часть 3 предназначена в основном для разработчиков, но также представляет некоторый интерес и для заказчика.

1.2. Область применения

[Примечание для студентов. Какие аспекты программы этот документ должен охватить?]

Этот документ охватывает требования к версии 0.01 игры *Встреча*. По данному документу будут делаться замечания относительно некоторых конкретных особенностей будущих версий. Цель этого — направлять процесс проектирования во время разработки приложения.

1.3. Определения, термины и сокращения

Приводятся в табл. 3.7.

Таблица 3.7. Термины

Сокращение или термин	Определение
Живой	Игровой персонаж считается «живым», если у него есть хотя бы одна ненулевая характеристика
C-требование	Сводка требований к приложению, сформулированных в форме, понятной клиенту
D-требование	Сводка требований к приложению, сформулированных достаточно четко для использования программистами при проектировании и реализации. По возможности D-требования должны быть также понятны и клиенту
Встреча	Название этой программы; также встреча двух игровых персонажей в одной зоне (не обязательно «контакт», см. ниже)
Контакт	Взаимодействие между персонажами игры, обычно сказывается на персонажах

Сокращение или термин	Определение
РИ	Ролевая игра: игра, обычно компьютерная, в которой игроки принимают на себя роли персонажей
Ролевая игра	См. «РИ»
Видеоигра	Игра, в которую играют на компьютере

1.4. Ссылки

План управления конфигурациями программного обеспечения (SCMP) для игры *Встреча*, версия 1.0.

Архитектура программного обеспечения (SDD) для игры *Встреча*, версия 1.2.

План управления программным проектом (SPMP) для игры *Встреча*, версия 1.2

План контроля качества (SQAP) для игры *Встреча*, версия 1.0.

План пользовательской документации (SUDP) для игры *Встреча*, версия 1.0.

Документация по тестированию программного обеспечения (STD) для игры *Встреча*, версия 1.0.

1.5. Обзор

Преднамеренно пропущено.

[Примечание для студентов. Автор документа не считал необходимым заполнять этот раздел и планирует сделать обзор в разделе 2.]

2. Общее описание

[Примечание для студентов. Сделайте эту часть достаточно общей, чтобы она не сильно изменялась в будущих версиях. Избегайте утверждений, повторяющихся в более поздних разделах.]

Встреча планируется как ролевая игра, моделирующая все стороны жизни главного персонажа игрока. Она должна быть интересна как мужчинам, так и женщинам. Оценка успеха игры *Встреча* остается за игроком. Обычно успех будет измеряться максимальным *числом очков-жизней*, набранным игроком, или возможностью игрока жить как можно дольше.

Некоторые игровые персонажи будут находиться под контролем игрока. Остальные, называемые *внешними*, будут управляться приложением. Игровые персонажи будут иметь фиксированное общее количество очков-жизней, распределенное среди всех характеристик, таких как сила, выносливость, терпение и т. д. Персонажи встречаются каждый раз, когда они одновременно попадают в одну зону, и могут затем вступать в контакт друг с другом. Результат контакта зависит от значений характеристик и от окружения, в котором контакт произошел. Контакт не обязательно должен представлять собой сражение или соперничество. Игроки имеют ограниченные возможности по перераспределению характеристик своих персонажей. Один из персонажей, контролируемых игроком, будет называться *главным персонажем игрока*.

В ранних версиях этой игры будет только один персонаж, контролируемый игроком, и один внешний персонаж.

Конечная природа персонажей будет определена согласно информации, полученной в опросах. Ожидается, что в первых версиях не будет анимации.

Игра *Встреча* должна, в конце концов, иметь высокий уровень настройки, так что пользователи смогут либо начинать заранее определенные игры, замещать заранее определенные персонажи, либо придумывать свои собственные персонажи и правила контакта.

Архитектура должна обеспечивать возможность расширения, включая версию игры через Интернет для нескольких людей.

2.1. Перспективы продукта

[Примечание для студентов. В этом разделе Встреча сравнивается с другими похожими или конкурирующими продуктами. Это полезный способ предоставить перспективы программы. Подзаголовок 2.1.1 этого раздела был изменен из стандарта IEEE в целях соответствия «концепции операций».]

Встреча должна удовлетворить нужды программистов в обладании большим влиянием на содержание видеоигры с дополнительным программированием. Она также нацелена на так называемую зрелую клиентуру. *Встреча* должна быть интересна как мужчинам, так и женщинам. Проект и документация игры *Встреча* позволит легко расширять и изменять игру. Допускается, что *Встреча* будет использоваться как первый шаг в создании таких программ, как моделирование офисного общения.

2.1.1. Концепции операций

[Примечание для студентов. Этот раздел дает общее представление о приложении с помощью тех средств, которые наиболее подходят для этого. В случае игры Встреча разработчики требований решили, что переходы состояний лучше всего покажут сущность игры.]

Игра *Встреча* может находиться в одном из следующих состояний (рис. 3.23):

- *Настройка*: состояние, в котором игрок делает начальную настройку в игре.
- *Оповещение*: система показывает окно с изображением статуса персонажей игрока.
- *Установка характеристик*: установка характеристик персонажа игрока. Этот процесс может длиться произвольное время и может производиться до тех пор, пока в окрестности персонажа игрока не появится внешний персонаж.
- *Контакт*: состояние, имеющее место каждый раз, когда внешний персонаж и главный персонаж игрока оказываются в одной зоне одновременно.
- *Ожидание*: игрок и внешний персонаж не активны.
- Эта схема переходов состояний прошла комплексное тестирование (прилагается).

2.1.2. Концепции пользовательского интерфейса

[Примечание для студентов. Следующие рисунки являются лишь набросками ключевого пользовательского интерфейса и используются для общего представления продукта. Весь пользовательский интерфейс определен подробно в разделе 3.

Мы изменили заголовок из стандарта IEEE («Пользовательские интерфейсы»), чтобы подчеркнуть, что это не детальное описание пользовательского интерфейса.]

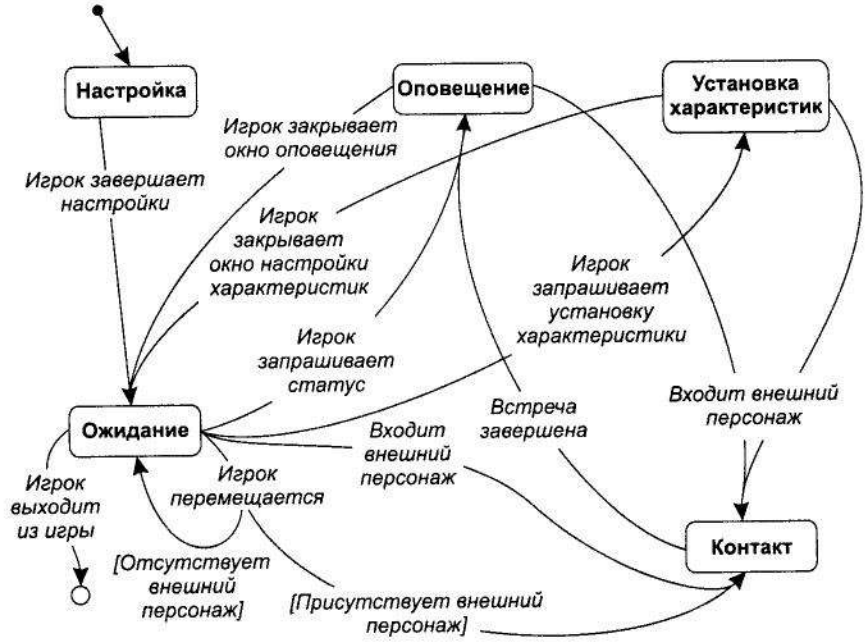


Рис. 3.23. Диаграмма переходов состояний для игры Встреча

2.1.2.1. Концепция зоны в пользовательском интерфейсе

Зоны, в которых происходят встречи, будут выглядеть весьма условно (рис. 3.24).

2.1.2.2. Концепция пользовательского интерфейса для установки значений характеристик

При установки значений характеристик при таком управлении, игроку показывается интерфейс в форме, набросанной на рис. 3.25. Линейка прокрутки используется для определения текущей характеристики, а текстовое окно для установки значения.

2.1.3. Аппаратные интерфейсы

Нет. В будущих версиях будет использоваться джойстик.

2.1.4. Программные интерфейсы

Нет.

2.1.5. Коммуникационные интерфейсы

Нет. В будущих версиях будет интерфейс для выхода в Интернет через модем.

2.1.6. Ограничения по памяти

Для игры *Встреча* потребуется не более 16 Мбайт оперативной памяти и 20 Мбайт вспомогательного запоминающего устройства (см. план теста; ссылка на тест будет приложена).

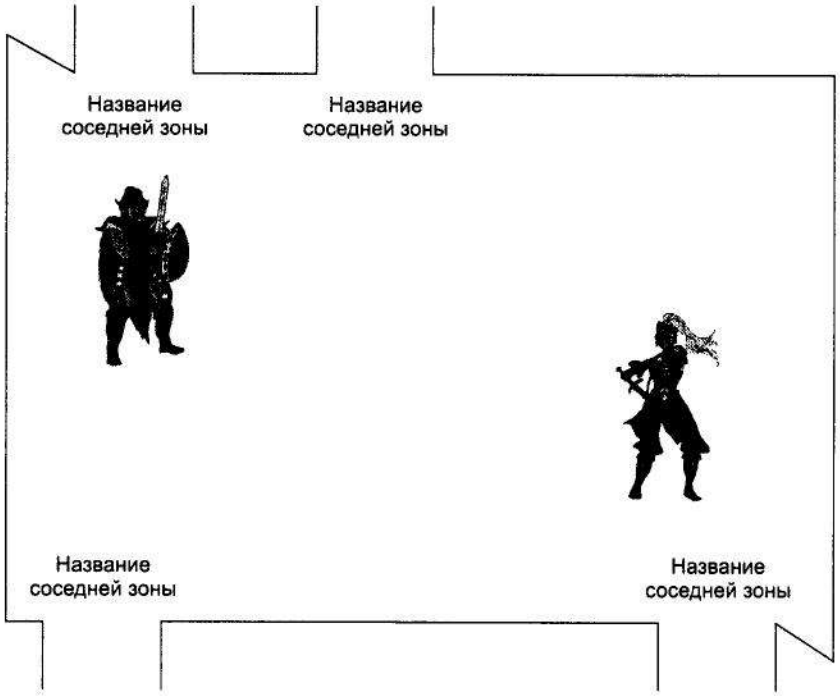


Рис. 3.24. Примерный снимок экрана игры Встреча

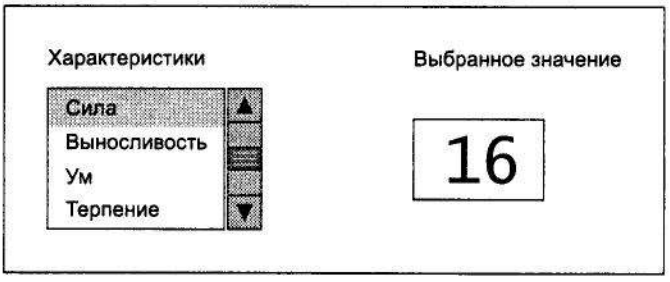


Рис. 3.25. Примерный набросок пользовательского интерфейса для установки значений характеристик персонажа

2.1.7. Операции

[Примечание для студентов. Обычные и особенные операции, требуемые от пользователя.]

[Будущие версии] Должна быть обеспечена возможность сохранять и загружать игру.

2.1.8. Требования по адаптации

[Примечание для студентов. Требования к выполнению на конкретном компьютере; версии на разных языках (например, французский, японский, испанский).]

Нет.

2.2. Функции продукта

[Примечание для студентов. Это сводка сведений об основных функциях приложения: более подробная, чем в разделе 1.5; менее подробная, чем в разделе 3. Составители этой спецификации программных требований решили, что варианты использования являются подходящим способом определения общей функциональности игры Встреча.]

В этом разделе определяется обязательная общая функциональность приложения, однако, целью не является предоставление полной спецификации. В разделе 3 представлены подробные требования.

2.2.1. Вариант использования «Инициализировать»

Действующее лицо: игрок игры *Встреча*.

Вариант использования: текст варианта использования «Инициализировать» приведен на рис. 3.26. Вариант использования показан в контексте варианта использования «Встретить внешний персонаж» и «Установить правила». «Инициализировать» — типичная последовательность, выполняемая игроками в начале игровой сессии.

Этот вариант использования соответствует тесту (ссылка на тест будет приложена) в документации тестов программы.

Рис. 3.26. Вариант использования «Инициализировать» в игре Встреча

2.2.2. Вариант использования «Перейти в соседнюю зону»

Действующее лицо: игрок игры *Встреча*.

Вариант использования:

1. *Игрок* щелкает на гиперссылке, соединяющей показанную зону с соседней.
2. *Система* показывает соответствующую соседнюю зону вместе с персонажем игрока.

2.2.3. Вариант использования «Встретить внешний персонаж»

Действующее лицо: игрок *Встречи*.

Вариант использования:

1. *Система* помещает внешний персонаж в зону нахождения персонажа игрока, или *Игрок* попадает в зону с внешним персонажем.
2. *Система* осуществляет контакт двух персонажей.
3. *Система* показывает результат контакта.
4. Если персонаж игрока или внешний персонаж имеет 0 очков-жизней, игра заканчивается.
5. В противном случае *Система* перемещает персонаж игрока в произвольную зону, отличную от той, где произошел контакт, и показывает его там.

2.3. Пользовательские характеристики

[Примечание для студентов. Покажите, какие люди будут типичными пользователями программы. Например: новичок, профессионал в программном обеспечении, бухгалтер с 5-летним компьютерным стажем и т. д.]

Ожидается, что пользователю будет 20–35 лет.

2.4. Ограничения

[Примечание для студентов. Все условия, которые могут ограничить возможности разработчика. Могут исходить из разных источников.]

Встреча будет работать на ПК с Windows 95 или более поздней с минимальной скоростью 100 МГерц. Языком разработки будет Java.

2.5. Предположения и зависимости

[Примечание для студентов. Могут быть сделаны любые допущения.]

Нет.

2.6. Распределение требований

[Примечание для студентов. Порядок, в котором требования будут выполняться.]

Требования, описанные в разделах 1 и 2 этого документа будут называться «С-требования», в разделе 3 — «D-требования». Основной аудиторией С-требований будет сообщество заказчиков, вторичной — разработчиков. Для D-требований ситуация обратная. Эти два уровня требований должны быть согласованными. Несогласованности должны быть отмечены отдельно как дефекты. В случае, когда требование сформулировано в С-требованиях и D-требованиях, приложение будет разрабатываться согласно D-требованиям, поскольку они более подробны.

Основные требования (упомянутые в разделе 3) должны быть реализованы в этой версии игры *Встреча*. *Желательные* требования должны быть по возможности осуществлены в этой версии, но не обязательны для разработчиков. Желательно, чтобы часть из них присутствовала в будущей версии. *Необязательные* требования будут добавлены по желанию разработчиков.

Глава 4
Анализ требований. Завершение SRS: добавление детальных требований

Детальные требования (D-требования) — единственный документ, в котором определяется конкретная природа программы. Уровень детализации должен быть полным, но не чрезмерным.

Содержание этой главы в контексте процесса разработки программы показано на рис. 4.1.

- Основы: разделы 4.1–4.5
- Детали: разделы 4.6–4.10
- Руководство по учебному проекту: Определение D-требований для игры *Встреча*
- Упражнения
- Пример: Спецификация требований к программному обеспечению (SRS), часть 2

Учебные цели этой главы заключаются в следующем.

- Овладеть разными вариантами организации D-требований:
  - по классам;
  - по вариантам использования;
  - по характеристикам;
  - по событиям.
- Суметь выполнить требования:
  - соблюсти достаточную степень детализации, чтобы суметь завершить проектирование и реализацию;
  - уметь выражать нефункциональные требования, например производительность.

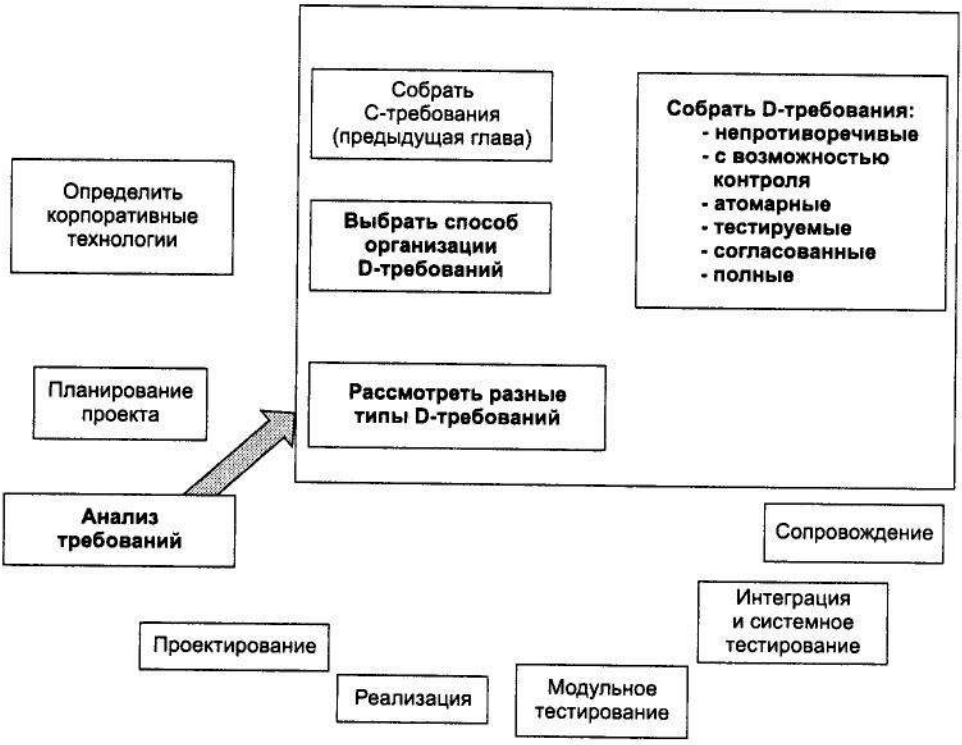


Рис. 4.1. Схема процессов разработки программ: темы главы

Основы

4.1. Введение в детальные требования (D-требования)

4.1.1. Значение детальных требований

Разработчикам программного обеспечения нужна база для проектирования и разработки. Эта база состоит из *детальных требований*. Их также называют *конкретными требованиями*, *функциональными спецификациями*, *требованиями разработчика* или *D-требованиями*. D-требования состоят из полного списка конкретных свойств и функциональности, которую должна иметь программа, сформулированных в подробностях. Каждое из этих требований пронумеровано, помечено и отслеживается по ходу разработки. D-требования должны быть согласованы с C-требованиями.

Предполагается, что D-требования будут читать преимущественно разработчики. Заказчики также заинтересованы в них и обычно могут понять или прокомментировать большинство из них. Вспомните, что основная аудитория для C-требований состоит из заказчиков.

Когда дело доходит до разработки программы, становится ясно, что «дьявол кроется в деталях». Например, в 1999 году NASA потеряла спутник стоимостью в сотни миллионов долларов из-за того, что данные, которые были восприняты в метрической системе, на самом деле были записаны в другой [14]. Самое удивительное здесь в том, что этот дефект был обнаружен лишь в день катастрофы. Как жаль, что он не был обнаружен на стадии разработки! На первой линии защиты от искажения или пропуска деталей стоят D-требования. Формулирование *всех* требований со всеми деталями — далеко не тупое занятие, как это может показаться на первый взгляд — включает в себя сложную задачу организации людей и документации. Чтобы понять эту задачу, представьте себе проблему организации документа требований в 20 томах так, чтобы, например, инженер NASA точно знал, где искать какое-то конкретное требование.

4.1.2. Типичная схема процесса анализа D-требований

Типичная последовательность действий для сбора и документирования D-требований показана на рис. 4.2. В разделе 4.5 описаны способы организаций детальных требований. D-требования создаются из C-требований, как показано в разделе 4.3. В идеале мы начинаем писать тесты для каждого из детальных требований одновременно с написанием самих требований. Хотя D-требования пишутся преимущественно для разработчиков, требования и их тесты просматриваются также и заказчиком. Затем D-требования должны быть проверены и выпущены (раздел 4.6.3).

Рис. 4.2. Схема процессов составления D-требований

4.2. Типы D-требований

Существуют несколько типов требований.

1. Функциональные требования:
   - функциональность приложения.
2. Нефункциональные требования.
   1) Производительность:
      - скорость;
      - пропускная способность (трафик);
      - использование памяти (оперативная память, жесткий диск).
   2) Надежность и доступность.
   3) Обработка ошибок.
   4) Интерфейсные требования.

      Как программа взаимодействует с пользователем и с другими программами.
   5) Ограничения:
      - точность;
      - ограничения на инструменты и язык, например «должен использоваться Fortran 88»;
      - ограничения проектирования;
      - стандарты, которые должны быть использованы;
      - платформы, которые должны быть использованы.
3. Обратные требования.

   Чего программа не делает.

Эта классификация применима как к C-, так и к D-требованиям. Во время написания C-требований эти различия часто занимают второе место после формирования основных идей заказчика относительно программы в целом. Однако классификация становится более важной при написании D-требований, поскольку она руководит процессами разработки и тестирования в разных аспектах. Приведенная классификация представляет собой модификацию схемы [92]. Далее в этой главе описаны разные типы требований.

4.2.1. Функциональные требования

Функциональные требования определяют работу, которую должно выполнять программное приложение (например, «приложение будет вычислять стоимость портфеля акций пользователя»). С другой стороны, требование, такое как «приложение будет выполнять вычисление стоимости каждого портфеля акций менее чем за одну секунду» не является функциональным, поскольку оно не определяет конкретную работу. Вместо этого оно оценивает работу или работы (определяет некоторое утверждение о работе).

4.2.2. Нефункциональные требования: требования к производительности

Требования к производительности определяют временные ограничения, которые должны быть выполнены в программе. Заказчики и разработчики обсуждают ограничения по времени вычислений, использование оперативной памяти, использование вспомогательных запоминающих устройств и т. д. Например:

Для любой балки Анализатор давления должен создать отчет типа 5 о давлении менее чем за минуту.

Требования к производительности являются важной частью приложений, работающих в реальном времени, в которых действия должны уложиться в предопределенные временные рамки. Примерами приложений реального времени могут быть программы избежания столкновений, контроля полетов и антиблокировочной тормозной системы.

4.2.3. Нефункциональные требования: надежность и доступность

Требования надежности определяют надежность в измеряемых величинах. Требования такого типа предполагают вероятность неидеальной работы программы и ограничивают область ее несовершенства. Например:

Приложение, управляющее радарами аэропорта, должно давать не более двух ошибок в месяц.

Доступность, близкая по смыслу надежности, оценивает степень, в которой приложение должно быть доступно пользователям. Например:

Приложение, управляющее радарами аэропорта, должно быть доступно на уровне один или два постоянно как на основном, так и на запасном компьютере. Оно может быть недоступно на одном из этих компьютеров на уровне один или два не более 2% времени в любой 30-дневный период.

4.2.4. Нефункциональные требования: обработка ошибок

Эта категория требований объясняет, как программа должна реагировать на возникающие ошибки. Например, что должна делать программа, если она получает сообщение из другой программы в неразрешенном формате? Это не касается ошибок, генерируемых самой программой.

В некоторых случаях *обработка ошибок* относится к действиям, которые должна предпринять программа, если она *сама* создала ошибку из-за дефекта в своей конструкции. Этот тип требований к ошибкам должен быть использован выборочно, поскольку нашей целью является создание безошибочных программ, а не исправление наших ошибок с помощью бесконечного кода обработки ошибок.

Например, предположим, что нам нужно определить требования для прибора, который автоматически определяет дозы лекарств для внутривенного введения

больному. Мы делаем предположение, что программа будет тщательно определена, разработана, реализована и проверена с тем, чтобы состав лекарств и дозы были корректными. Однако было бы правильно в таком случае определить независимую проверку состава и дозы лекарств перед его выпуском в кровь и соответствующим образом определить обработку ошибок.

В итоге, проверка ошибок в самой программе уместна только для критичных частей программы.

4.2.5. Нефункциональные требования: интерфейсные требования

Интерфейсные требования описывают формат, в котором программа общается с окружением.

Стоимость посылки статьи от адресата получателю должна постоянно показываться в текстовом окне «Цена».

Формат, используемый для передачи сообщений «Ожидаемая статья» для взаимодействия с почтовыми кампаниями, будет представлять собой строку вида exp<source>, где <source> — это строка из Таблицы стандартов городов.

В первом примере показано требование пользователей программы. Второе предоставляет формат сообщения для обмена информацией с другими программами. Оба требования являются интерфейсными.

4.2.6. Нефункциональные требования: ограничения

Ограничения на проектирование или реализацию описывает границы или условия того, как приложение должно быть задумано и разработано. Эти требования не должны использоваться вместо процесса проектирования — они всего лишь определяют условия, наложенные на проект заказчиком, а также окружение или другие обстоятельства. Например, сюда можно отнести точность:

Вычисления оценки ДТП системой AEF должны быть выполнены с точностью до одного сантиметра.

Часто также накладываются ограничения по инструментам и языкам. Сюда относятся исторически сложившиеся традиции организации, совместимость и опыт программистов.

Система AEF должна быть разработана на Fortran 88.

Ограничения на проектирование накладываются в связи с тем, что этого могут требовать финансово заинтересованные в проекте лица. Такие условия ограничивают свободу проектирования разработчиков, например:

Система AEF должна использовать систему UCF для демонстрации результатов столкновения.

Ограничение, требующее следовать определенному стандарту, часто определяется политикой фирмы или заказчиком.

Документация AEF должна удовлетворять требованиям Федерального стандарта 1234.56.

Код AEF должен быть документирован с использованием руководства кампании для документирования кода, версия 5.2.

Проекты часто ограничены платформами, на которых они будут использоваться. Например:

Система AEF должна работать на компьютерах Ajax 999 модель 12345 со 128 Мбайт оперативной памяти и 12 Гбайт дискового пространства.

4.2.7. Обратные требования

Обратные требования определяют, чего программа *не будет* делать. Логично, что существует бесконечное число обратных требований: мы выбираем только те, которые разъясняют возможное непонимание. Например:

Система AEF (см. предыдущий раздел) не обязательно должна анализировать данные ДТП.

4.2.8. Отображение типов требований на стандарт IEEE 830-1993

Основные разделы стандарта IEEE 830-1993 с их отображением на типы требований, приведенные выше, перечислены на рис. 4.3.

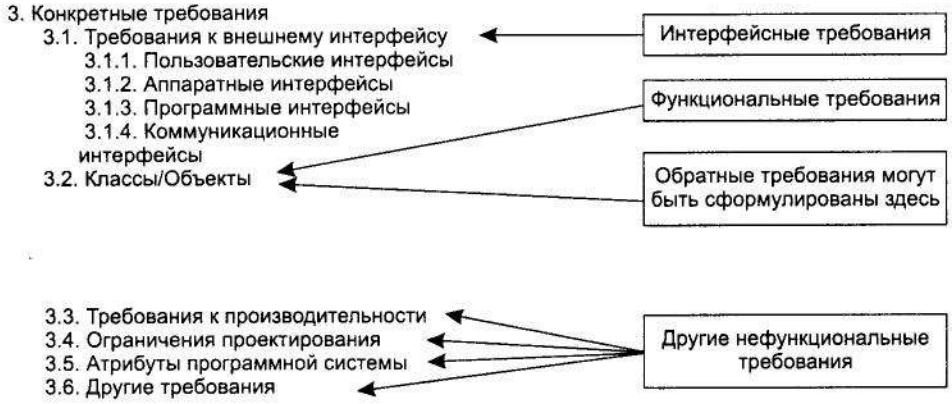


Рис. 4.3. Организация SRS согласно стандарту IEEE 830-1994: детальные требования с объектно-ориентированной организацией

4.3. Желательные свойства D-требований

Мы хотели бы подчеркнуть значительное влияние, которое может оказать на проект пропуск тех или иных деталей. Чтобы помочь убедиться, что все детали были охвачены, мы определим свойства, которыми должны обладать D-требования.

В частности, D-требования должны быть полными и согласованными. Нужно, чтобы можно было отследить каждое требование в проекте, протестировать его работу и разработать его согласно его приоритету. Мы будем исследовать требования на наличие этих характеристик, когда будем оценивать качество (раздел 4.6). В конце данного раздела представлена последовательность шагов, которая может быть использована для выражения подробных требований.

4.3.1. Прослеживание

4.3.1.1. Прослеживание функциональных требований

Представьте себе программу с 1000 детальных требований. Без возможности четкого контроля каждого требования от проекта программы до самого кода, реализующего это требование, было бы сложно убедиться в том, что программа разработана в соответствии с требованиями. Когда требования меняются (чего следует ожидать), это становится еще сложнее. Возможность отображать каждое требование на соответствующие части проекта и программы называется *прослеживанием*. Один из способов, помогающих достичь этого, заключается в отображении каждого функционального D-требования на конкретную функцию целевого языка. Эта техника использована в примере (рис. 4.4): показаны части проекта, которые должны быть объединены для возможности контроля. Достижение и поддержка этого уровня контроля во время разработки представляет собой сложную задачу.

В качестве примера представьте себе такое требование к игре *Встреча*.

Когда внешний игровой персонаж появляется в зоне, содержащей главный персонаж игрока, или наоборот, эти персонажи вступают в контакт друг с другом.

Значение этого утверждения очевидно, однако нужно понимать, какая часть проекта и кода будет отвечать за выполнение этого требования. При использовании объектно-ориентированного подхода мы можем связать это требование с конкретной функцией конкретного класса. Вопрос о том, какой класс отвечает за эту функцию, нетривиален, и он возникает постоянно при использовании объектно-ориентированного подхода. Что касается приведенного выше примера, объекты *Зона* должны иметь возможность распознавать контакт, поскольку именно эти объекты предположительно должны владеть информацией о находящихся «в них» персонажах. В частности, требование можно будет отследить с помощью определенного кода, обрабатывающего события для класса *Зона*.

По мере продвижения проекта документ с требованиями должен поддерживаться согласованным с архитектурой и реализацией. Когда требования сложно отследить в архитектуре и коде, разработчики часто стараются избежать обновления документа требований при внесении изменений в исходный код, поскольку для этого потребуется много усилий. В конце концов, такая порча документа приводит к увеличению работ и затрат на проект. Это явление проиллюстрировано в нижеследующем примере.

1. Разработчику Биллу поручили сделать изменения в программе. Биллу кажется, что привести документ с требованиями в соответствие с кодом сложно, и он решает не делать этого.

2. Разработчику Джону поручают сделать новые изменения. Он делает изменения в коде, тестирует их и начинает обновлять документ с требованиями. Но все говорят ему не беспокоиться об этом, поскольку документ устарел в некоторых местах и никто ему уже не доверяет. Джону говорят, что нет смысла делать это, поскольку документ все равно никто читать не будет. Так что Джон бросает это занятие и продолжает свое программирование. Таким образом, расхождения между документом и кодом увеличиваются.

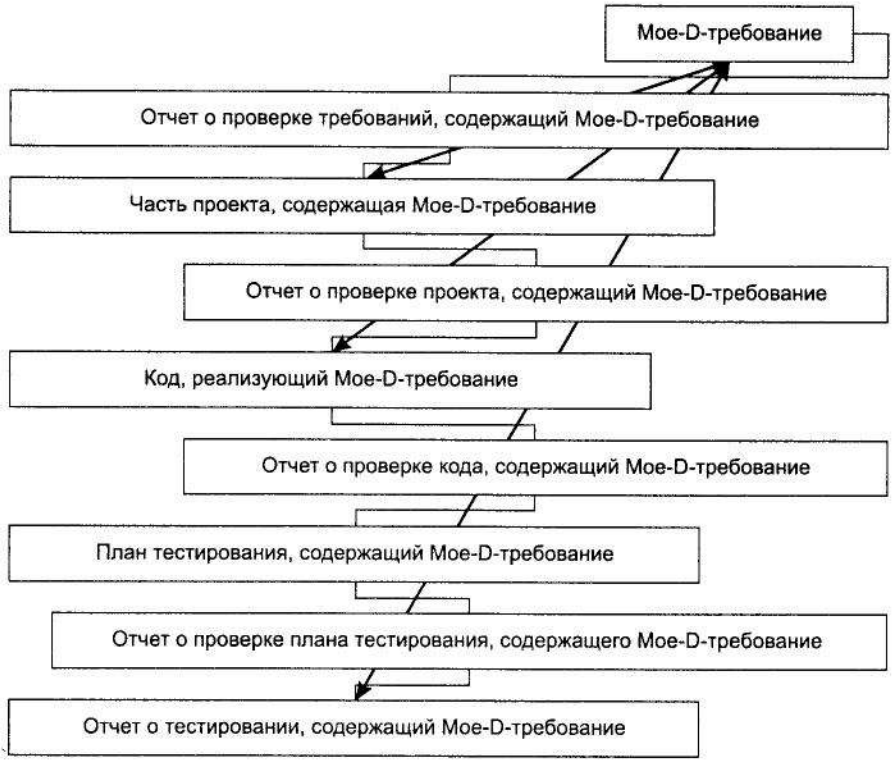


Рис. 4.4. Контроль D-требования (стрелки показывают основные линии прослеживания)

Даже самые добросовестные разработчики игнорируют необходимость обновления документа требований, если документ в целом считается ненадежным. С другой стороны, если документы четко и просто ссылаются друг на друга и руководство делает документирование обязательным, разработчики поддерживают документы в очень хорошей профессиональной форме. Другими словами, система, используемая для поддержания соответствия D-требований с проектом и соответствующим кодом, должна быть очень четкой и конкретной.

Когда код, осуществляющий требование, находится в нескольких местах, прослеживание достигается посредством использования *матрицы прослеживания требований*, пример которой приведен в табл. 4.1.

Таблица 4.1. Матрица прослеживания требований

Требование	Модуль 1	Модуль 2	Модуль 3
1783	showName()	computeBal()	getInterest()
1784	showAccount()	showAddress()	showName()

Как показано в табл. 4.1, требование 1783 реализовано посредством функций `showName()` в модуле 1, `computeBal()` в модуле 2 и `getInterest()` в модуле 3. Изменение в этом требовании повлечет изменение в одной или нескольких из этих функций. Это следует четко контролировать, поскольку эти функции могут участвовать также и в выполнении других требований (например, `showName()` используется еще и для реализации требования 1784). В результате изменения, сделанные для удовлетворения одного требования, могут нарушить другое. Поскольку с отношениями «многие-ко-многим» трудно работать, мы стараемся сделать отображение между требованием и функцией «один-к-одному».

Мы хотим, чтобы каждое D-требование можно было проследить *прямо* и *обратно*. Предыдущие обсуждения касались прямого отслеживания от D-требований до исполнения. Обратное отслеживание D-требований означает, что требование представляет собой четкую последовательность одного или нескольких C-требований. Например, D-требование:

внешний персонаж должен двигаться от зоны к зоне со средним интервалом в пять секунд

может быть прослежено назад к следующему C-требованию, которое было частью раздела 2.0 в SRS:

остальные [персонажи], называемые внешними, должны контролироваться программой.

Такой обратный контроль является основой проверки D-требований.

Возможность полного прослеживания означает, что каждое D-требование связано с конкретным элементом проекта, а также с тестом элемента (рис. 4.5). Рисунок показывает преимущество жесткого соответствия между каждым отдельным функциональным требованием, описывающей требование частью проекта и реализующей его частью кода. Они связаны с тестом требования (тесты являются темой главы 8).

4.3.1.2. Контроль нефункциональных требований

Предыдущее обсуждение относилось к функциональным требованиям, однако как же мы прослеживаем *нефункциональные* требования? Это может оказаться сложной задачей, поскольку одному нефункциональному требованию может соответствовать более одной части проектирования и реализации. Например, требование того, чтобы каждый *контакт* происходил менее чем за одну секунду, может затронуть код класса *Контакт* и (или) класса *ПерсонажИгры* и (или) класса *Зона*. Наша задача на данном этапе заключается в определении нефункциональных требований возможно более четким языком. Чтобы прояснить нефункциональные требования, мы также затронем вопросы проектирования и реализации.

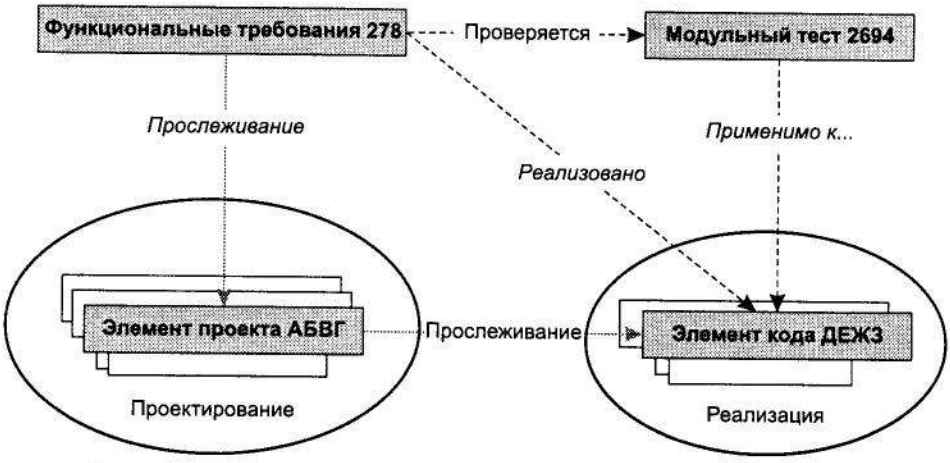


Рис. 4.5. Прослеживание и тестирование функциональных D-требований

Одна из целей стадии проектирования — изолировать каждое нефункциональное требование в отдельный элемент проекта. В случае требований к производительности делается попытка изолировать самые медленные вычислительные компоненты. Каждую функцию, связанную с требованиями к производительности, сопровождают соответствующие проверяемые нефункциональные комментарии. Желательно, чтобы эти требования были количественными, как, например, «нужно завершить работу менее чем за одну миллисекунду в худшем случае». Аналогично, в случаях, когда определены ограничения по памяти, мы выделяем функции, больше других расходующие память.

Исследования показали, что основную часть работы в программе выполняет относительно небольшая часть функций, поэтому поиск нескольких основных расходующих время функций может принести хорошие результаты. Вернемся к требованию выполнения контакта в игре *Встреча* за одну секунду, которое уже упоминалось. Во время проектирования и реализации мы ищем типичные компоненты, расходующие время при обработке контакта. К таким компонентам относятся циклы, отображение графики, а также сетевая передача данных. Циклы и передачи данных не связаны с обработкой контакта, поэтому проводится тест. Тест должен убедить нас в том, что графика и графический пользовательский интерфейс, требуемые при контакте, работают достаточно быстро. Функция, расходующая большую часть времени, — это, вероятно, либо функция «Вступить в контакт с внешним персонажем» класса *Контакт*, либо функция вывода результатов контакта.

Чтобы утвердить нефункциональные требования, нам пришлось связать каждое из них с планом тестирования — желательно делать это во время написания требования. Типичная связь функциональных и нефункциональных требований с реализацией и тестированием, обсуждавшимися выше, представлена на рис. 4.6. На рисунке проиллюстрирован случай, когда нефункциональным требованиям может соответствовать несколько компонентов, а также когда для утверждения нефункциональных требований необходимо системное или интегральное тестирование, поскольку их верификация (то есть проверка до запуска программы) может оказаться весьма сложной.

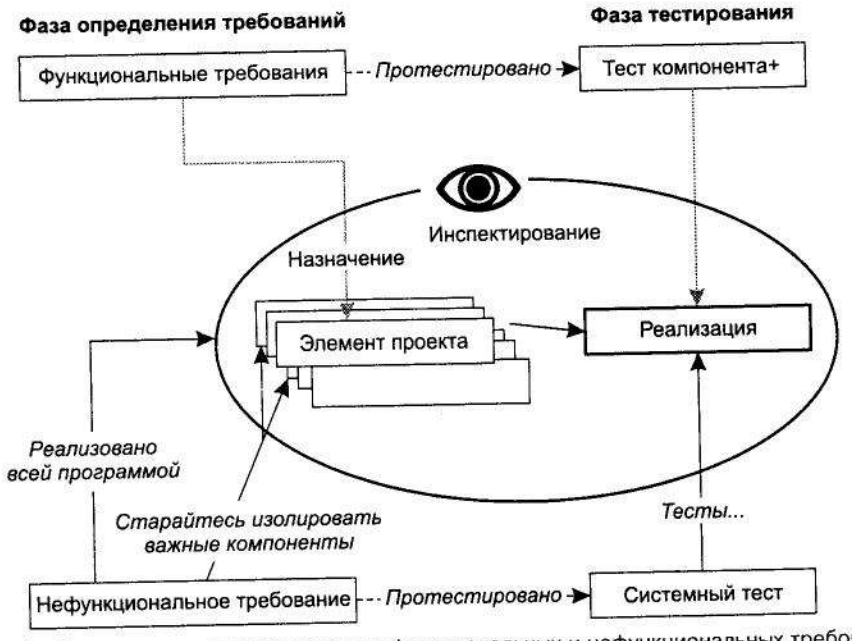


Рис. 4.6. Прослеживание и тестирование функциональных и нефункциональных требований

4.3.2. Пригодность к тестированию и однозначность

Должна существовать возможность валидировать требование, убедившись, что требование было корректно реализовано. (Вспомните, что в главе 1 *валидация* относится к проверке того, что артефакт функционирует так, как должен.) Требования, которые можно протестировать, называются *тестируемыми*. Нетестируемые «требования» не имеют большой практической ценности. Приведем пример нетестируемого требования и перечислим действия, которые нужно выполнить для того, чтобы требование стало тестируемым.

Система должна показывать разницу в зарплате между зарплатой клиента и средней зарплатой в мире для той же специальности.

Это требование не протестировать, поскольку упоминаемую среднюю зарплату невозможно определить (несмотря на то, что она существует).

Улучшенная версия требования:

Система будет показывать разницу в зарплате между зарплатой клиента и предполагаемой средней зарплатой для той же специальности согласно цифрам, опубликованным на веб-сайте ООН на время запроса.

Если D-требование записано нечетко или двусмысленно, мы не сможем определить, правильно оно было реализовано или нет. Пример неоднозначного требования и его исправленная версия приведены ниже.

Игрок может определять значения характеристик персонажей игры.

В любое время? Вероятно, нет. Придется протестировать разные случаи, с соответствующими ненужными затратами, и мы получим неправильный результат.

Улучшенная версия этого требования.

Каждый раз, когда внешние персонажи отсутствуют в зоне нахождения главного персонажа игрока, игрок может изменить значения характеристик персонажа, сохранив сумму значений неизменной. Для этого используется окно характеристик игрока (см. раздел 3.2.ПИ в конце главы).

4.3.3. Приоритет

Часто бывает трудно реализовать всю запланированную функциональность программы в срок и не выходя за рамки бюджета. Как упоминалось в разделе 2.3.1, могут измениться *возможности*, *сроки*, *уровень качества* и *стоимость*. Поэтому, если сроки, бюджет или уровень качества менять нельзя, единственной альтернативой остается изменение возможностей, то есть уменьшение количества реализованных требований. Этот процесс отсеивания выполняется запланированным способом. Одна из техник — расставить приоритеты между детальными требованиями. Упорядочивание всех требований обычно является пустой тратой времени — вместо этого многие организации классифицируют требования по трем (иногда четырем) категориям. Мы назовем их *важные*, *желательные* и *необязательные*. Убедимся, что проект реализует все существенные требования. Использование трех категорий представляет собой применение отбраковки, упомянутой в главе 2. Назначение приоритетов требованиям сказывается на проектировании, поскольку желаемые и необязательные требования часто указывают направление, в котором должно двигаться приложение. Широко известная мудрость гласит, что предприятия получают порядка 80 % дохода от использования всего лишь 20 % требований. Поэтому, если приоритеты расставлены правильно (например, только около 20 % требований отнесены к существенным), можно получить максимальную прибыль от приложения с учетом требуемой работы. Этот полезный фактор следует учитывать, если проект начинает выпадать из временных рамок. Пример отбраковки требований:

[важно] Все игровые персонажи имеют одинаковые наборы характеристик.

[желательно] Каждая зона имеет набор предпочтительных характеристик.

[не обязательно] Персонаж игрока должен взрослеть с каждой встречей. Скорость взросления может быть установлена во время начальных настроек. По умолчанию она равна одному году за каждую встречу.

Приведенный далее пример содержит некоторые приоритетные D-требования для первой версии игры *Встреча*. Они перечислены здесь для того, чтобы читатель смог себе представить, с вопросами какого типа ему придется иметь дело. Некоторые желательные требования в будущих версиях перейдут в категорию важных. Требования приведены в черновом варианте и, очевидно, нуждаются в повторном упорядочивании. Они будут исправлены дальше в тексте главы и в примере.

Предварительный черновик D-требований игры Встреча

(Требования еще не организованы: см. исправленную форму в примере в конце главы.)

- (не проверено:)[важно] Каждый игровой персонаж в видеоигре *Встреча* должен иметь имя.
- (не проверено:)[важно] Каждый игровой персонаж имеет некоторый набор характеристик, значение каждой из которых представлено вещественным числом с дробной частью.
- (не проверено:)[важно] Игра происходит в зонах, каждая из которых соединена с другими зонами посредством выходов.
- (не проверено:)[важно] Каждый раз, когда персонаж игры входит в зону, в которой находится другой персонаж, и один из этих персонажей контролируется игроком, персонажи могут либо по выбору, либо принудительно вступать в контакт друг с другом.
- (не проверено:)[важно] Каждый раз, когда персонаж, контролируемый игроком, находится один в зоне, игрок может изменять значения его характеристик.
- (не проверено:)[желательно] Имя каждого персонажа в игре может состоять не более чем из 15 символов.
- (не проверено:)[желательно] В любой момент персонаж будет иметь некоторое число очков-жизней, под которым понимается сумма значений его характеристик.
- (не проверено:)[желательно] Каждая зона имеет набор предпочтительных характеристик.
- (не проверено:)[желательно] Зоны сражений требуют силы и выносливости; зоны жилых комнат требуют внимательности и ума.
- (не проверено:)[желательно] Сумма значений характеристик игрового персонажа, соответствующих требованиям зоны, будет называться значением персонажа в зоне. Во время контакта система будет сравнивать значения персонажей в зоне и передавать сильнейшему половину очков-жизней слабейшего. Например, представьте себе, что игрок вступает в контакт с внешним персонажем в зоне, требующей выносливости и внимательности, и p_s — значение выносливости игрока. Предположив, что $p_s + p_a > f_s + f_a$, получим:

  $$p'_s = p_s + f_s / 2, p'_a = p_a + f_a / 2, f'_s = f_s / 2, f'_a = f_a / 2,$$

  где штрихом отмечены значения после контакта.
- (не проверено:)[не обязательно] При контакте расчет результатов будет происходить менее чем за секунду.
- (не проверено:)[не обязательно] Персонаж игрока будет стареть с каждым контактом. Скорость взросления можно установить в начальных настройках. По умолчанию за каждый контакт добавляется год.

- (не проверено:)[не обязательно] Персонажи, управляемые игроком, теряют или набирают значения своих характеристик в конце каждого контакта со скоростью +2 % при значении возраста до 30 и –2 % при значении возраста больше 30.

Расстановка приоритетов требований связана с итерацией, реализующей их. Например, если мы не сможем реализовать необязательное требование

При контакте расчет результатов будет происходить менее чем за секунду

на второй итерации, оно может появиться на следующей итерации, но уже с более высоким приоритетом. Требования для каждой итерации хранятся в отдельном документе. Это помогает лучше понять более поздние требования.

4.3.4. Полнота

Мы стараемся сделать каждое подробное требование самодостаточным, но на практике это редко удается, когда требования часто ссылаются на другие требования. *Полнота* набора требований гарантирует, что не было сделано никаких пропусков, подрывающих сформулированные требования. Ниже представлен неполный набор требований. Без указания того, как должно отображаться видео, этот набор требований является неполным.

Начало требований.

1. *Приложение должно показывать фильм, название которого введено в строке приглашения.*
2. *Приложение должно показывать все фильмы, фамилия режиссера которых введена в строке приглашения.*
   - *Порядок показа определяется с помощью клавиши перемещения курсора* ↑.
3. *Приложение должно показывать все фильмы с участием актера, фамилия которого введена в строке приглашения.*
   - *Порядок показа определяется с помощью клавиши перемещения курсора* ↑.

Конец требований.

Опущено: Специфицировать, как «показать» видео!

Изучая пример видеоигры *Встреча*, попробуем критически оценить, имеем ли мы полный набор требований. Вероятно, хорошим способом оценить текущий статус требований будет просмотр их с помощью вариантов использования.

- Мы устанавливаем характеристики персонажа игрока в гардеробной. Мы можем проверить наличие всей необходимой функциональности и данных для этого.
- Мы переходим в соседнюю зону. Проверяем наличие всей необходимой функциональности для этого.
- Мы обнаруживаем внешний персонаж. Удостоверяемся, что все детали для этого предоставлены.

4.3.5. Состояние ошибки

Для каждого требования нужно поставить вопрос: «Что будет, если произойдет ошибка?». Рассмотрим пример, предложенный Майерсом [82], — требование, которому не хватает обязательных условий ошибки.

Функция, определяющая, образуют ли три числа равносторонний, равнобедренный или разносторонний треугольник.

Спецификация этого требования неполная, поскольку она не рассматривает состояний ошибки. Следующая версия является более полной. Отсутствие рассмотрений ошибок в спецификациях требований становится особенно заметным при тестировании функций, поскольку тестер предвидит ошибки и должен знать, что должно выводиться в этом случае.

Функция, определяющая, образуют ли три числа:

1 — равносторонний треугольник (со сторонами больше нуля и равными), в этом случае выводится слово «равносторонний»,

2 — равнобедренный треугольник (со сторонами больше нуля, ровно две из которых равны, и которые образуют треугольник), в этом случае система возвращает слово «равнобедренный»,

3 — разносторонний треугольник (стороны которого все больше нуля и образуют треугольник, не являющийся ни равносторонним, ни равнобедренным), в этом случае система возвращает слово «разносторонний»,

4 — треугольник не образуется, в этом случае система возвращает слово «нет».

Тщательный анализ требований имеет дело с *неразрешенным* вводом. Существует большой соблазн предположить, что пользовательский интерфейс для некоторого требования не разрешает ввод отрицательных значений, так что функции не придется работать с ошибочными данными. Вообще говоря, такое предположение неразумно, поскольку оно перекладывает требования корректности данных на *пользователей*. Это увеличивает зависимость между частями программы. Хотя хорошей практикой считается отслеживать некорректный ввод данных на уровне графического пользовательского интерфейса и не обязывать пользователя вводить только разрешенные значения, это не заменяет жестких требований в других местах. Автор рекомендует требовать отслеживания некорректных данных во многих, если не во всех, местах программы. Это один из эквивалентов давно сложившейся практики программирования, когда избыточность используется для повышения надежности.

4.3.6. Согласованность

Набор D-требований *согласован*, если между требованиями нет противоречий. По мере увеличения числа D-требований согласованность может стать труднодостижимой, что иллюстрирует следующий пример:

Требование 14. Игровые персонажи могут нести только основные продукты питания.

...

Требование 223. Любой персонаж несет воду.

...

Требование 497. Только хлеб, масло, молоко и соль являются основными продуктами питания.

Объектно-ориентированная организация требований помогает избежать несогласованности благодаря классификации D-требований по классам и с помощью разложения их на простейшие. Однако и это не гарантирует согласованности, поэтому следует проверять согласованность вместе с другими рассмотренными характеристиками.

4.3.7. Подведение итогов процесса написания детальных требований

Мы подводим итог процессу, которому можно следовать при выражении одного требования. Большинство шагов, вынесенных в отдельные пункты, были описаны в этом разделе как желательные характеристики требований.

Вот дополнительные замечания, соответствующие пронумерованным шагам во врезках «Один из способов...».

1. В разделе 4.5 обсуждаются способы организации детальных требований с акцентом на объектно-ориентированный стиль. Метод организации должен быть определен до написания D-требований.
2. Оценка того, является требование прослеживаемым или нет, эквивалентна представлению архитектуры программы и того способа, которым архитектура должна удовлетворять этому требованию. Проще всего это сделать, если требование четко соответствует методу.

ОДИН ИЗ СПОСОБОВ НАПИСАТЬ ДЕТАЛЬНОЕ ТРЕБОВАНИЕ

1. Классифицируйте требование как функциональное или нефункциональное:
   - используйте подсказки IEEE SRS для большинства нефункциональных требований;
   - выберите метод организации функциональных требований.
2. Аккуратно отсортируйте требования:
   - функциональное требование более-менее соответствует методу;
   - слишком большое — трудно управлять;
   - слишком маленькое — нет смысла рассматривать отдельно.
3. По возможности сделайте его прослеживаемым. Убедитесь в возможности прослеживания при проектировании и реализации.
4. Сделайте его тестируемым. Набросайте конкретный тест, устанавливающий выполнение требования.
5. Убедитесь в недвусмысленности требования. Убедитесь в очевидности замысла.
6. Назначьте требованию приоритет. Например, высокий («важно»), низкий («не обязательно») или средний («желательно»).
7. Проверьте полноту требования. Для каждого требования убедитесь в присутствии всех остальных необходимых сопутствующих требований.
8. Добавьте состояния ошибки:
   - сформулируйте, что конкретно требуется для нештатных ситуаций;
   - в критичных местах добавьте состояния ошибок программирования.
9. Проверьте согласованность. Убедитесь, что ни одно требование не противоречит каким-либо аспектам другого требования.

3. Проще наметить тест для требования во время написания самого требования. Это не только делает требование более понятным, но и определяет тестируемость требования.
4. Многие требования зависят от конкретных данных, и мы должны указывать, как требование должно работать в случае неправильных или несогласованных данных. Для критичных требований сюда также следует относить ошибки неправильного проектирования или программирования. Например, допустимое требование: «когда нажата кнопка On, высокоинтенсивный рентгеновский луч будет включен, если параметры удовлетворяют условиям». Недопустимое требование: «позиции при игре в крестики-нолики должны отображаться при условии, что ни один игрок не сделал на два хода больше другого».

4.4. Диаграммы последовательности

Диаграммы последовательности являются графическим представлением передачи управления и особенно полезны для визуализации реализации вариантов использования. Помимо использования их для анализа требований, как мы делаем в этой главе, мы также используем их более подробные версии для проектирования, как показано в главе 6.

Диаграммы последовательности заставляют нас рассуждать в терминах объектов. В такой диаграмме жизненный цикл каждого участвующего объекта показан вертикальной линией с именем объекта и указанием его класса вверху. Каждое взаимодействие между объектами отображается горизонтальной стрелкой от объекта, инициирующего взаимодействие, к объекту, выполняющему дальнейшие функции. Начало диаграммы последовательности для варианта использования *Встречи* показано на рис. 4.7. Далее приведены замечания к рис. 4.7.

1. Прежде всего, мы обозначаем объект, инициирующий вариант использования, согласно UML с помощью прямоугольника. Это объект класса *ИграВстреча*. Тот факт, что упоминается не конкретный объект, а весь класс, указывает на то, что либо объект не требуется вообще (например, используются статические методы), либо объекта без какого-либо конкретного имени («анонимного» объекта) будет вполне достаточно.
2. Узкий продолговатый прямоугольник обозначает выполнение функции объекта.
3. Затем мы показываем выполнение операции объекта *Зона*, инициированного из *ИграВстреча*: в этом случае используется операция создания, которая обычно является конструктором. Созданный объект — объект *гардеробнаяЗона*. Неожиданно мы принимаем решение сделать *Зону* классом. (Имя, которое мы выбрали для этого класса, не совсем корректно — слишком обобщенно.) Нам следовало назвать его *ЗонаВстреч*; в дальнейшем мы проследим последствия этого дефекта.
4. Обратите внимание, что жирная линия для объекта *гардероб* начинается лишь тогда, когда объект начинает существовать.

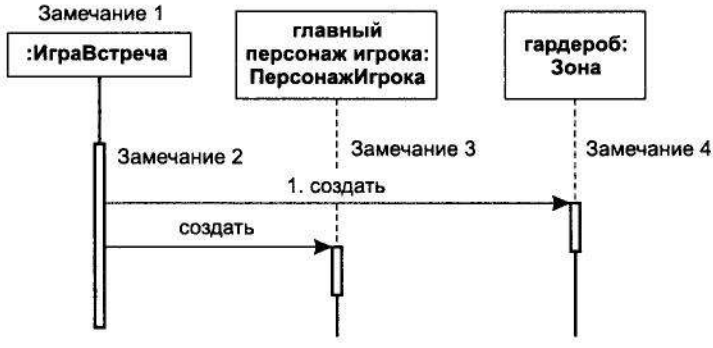


Рис. 4.7. Начало диаграммы последовательности для варианта использования «Инициализировать»

В законченной диаграмме последовательности (рис. 4.8) представлены операции. Эти операции показывают работу, инициируемую объектом с исходящей стрелкой и выполняемую объектом, на который указывает стрелка. Каждая операция обычно реализуется функцией на стадии проектирования. Цифрами на рис. 4.8 обозначены ссылки на варианты использования.

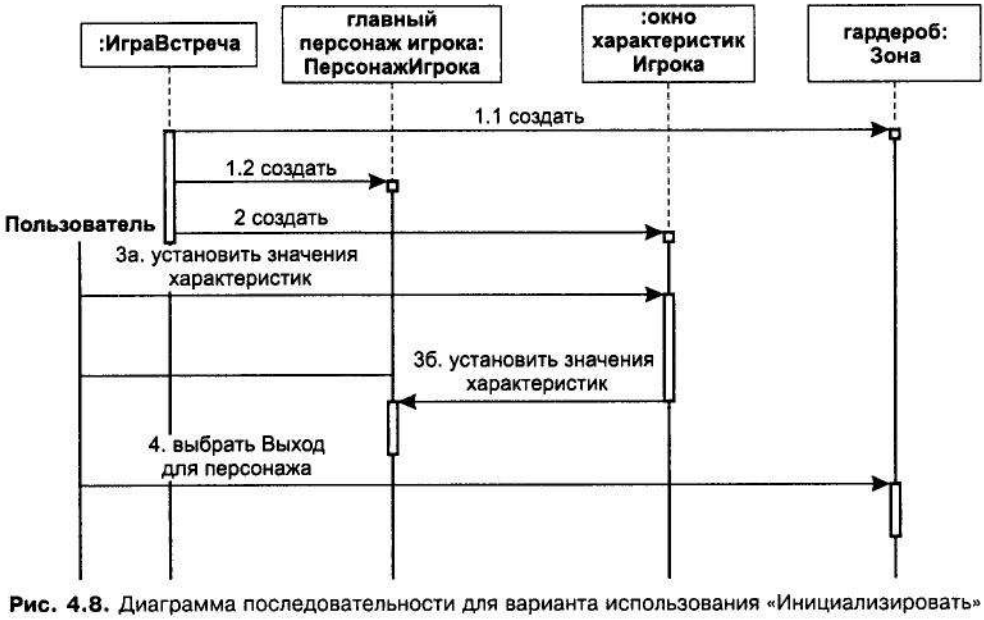


Рис. 4.8. Диаграмма последовательности для варианта использования «Инициализировать»

Полная диаграмма последовательности для варианта использования приведена на рис. 4.8. Идея состоит в том, что по мере того как вы проигрываете вариант использования, становится очевидной необходимость некоторых объектов. Эти объекты заставляют вас вводить новые классы. В варианте использования «Инициализировать» первое действие, предпринимаемое программой — показать гардероб и главный персонаж игрока. Нам нужен объект, который выполнял бы этот

процесс создания. Разумным решением этой задачи будет один объект класса *ИграВстреча*. *ИграВстреча* создает объект *гардероб* класса *Зона*, затем «говорит» объекту *гардероб* показать себя. Каждый вытянутый вертикально прямоугольник показывает выполнение вызванной функции. Некоторые из пронумерованных шагов варианта использования реализуются вызовом более чем одной функции. Мы представили класс *ПерсонажИгрока*, которому принадлежит только один объект — *главныйПерсонажИгрока*.

На этот момент диаграмма последовательности служит для определения основных классов. Обычно диаграмму последовательности приходится менять и уточнять во время проектирования. Примером является определение класса, отвечающего за создание объекта *гардероб*. Архитектура (глава 5) может вызвать создание объекта *гардероб* не из объекта класса *ИграВстреча*, а из самого класса *Зона* либо еще из какого-нибудь другого класса.

Диаграммы последовательности могут использоваться для определения параллельных потоков управления. Например, мы можем захотеть, чтобы персонажи игры двигались независимо от зоны к зоне. В нотации UML для этого случая предусмотрена стрелка от одной колонки к другой, причем изображается только нижняя половина стрелки (рис. 4.9). Это означает инициализацию потока управления в конце стрелки.

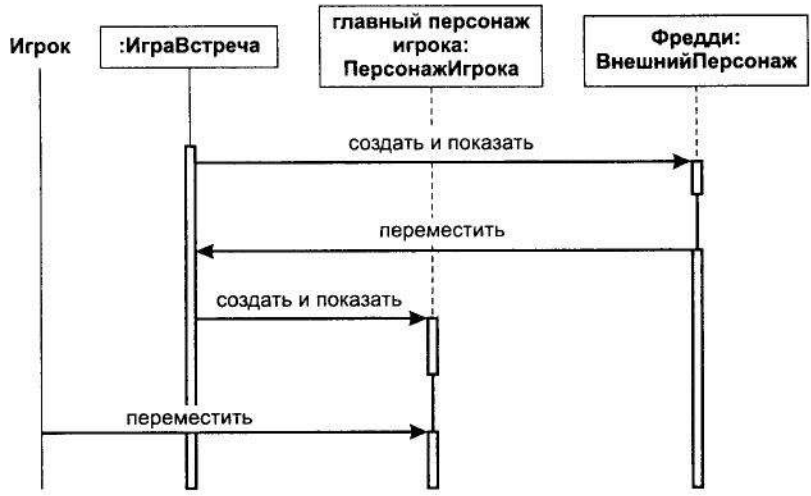


Рис. 4.9. Диаграмма последовательности, показывающая параллелизм

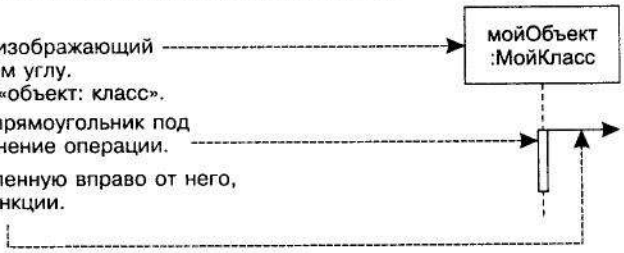
Параллельные вертикальные полосы уместно использовать для визуализации параллелизма. Однако диаграммы последовательности неудобно использовать для описания синхронизации. Это можно сделать с помощью *диаграмм деятельности* UML [64]. За исключением случаев обязательного выражения требований заказчика, такие вопросы, как синхронизация, могут быть отложены до стадии проектирования.

Подведем итог шагов, необходимых для создания диаграммы последовательности. В случае, когда первым объектом является пользователь, вместо прямо-

угольника будет достаточно простой метки вверху. Отметьте, что объект, отвечающий за выполнение отмеченной на стрелке работы, стоит *в конце* (а не в начале) стрелки. Наиболее значимой частью процесса является выбор классов. Мы должны решить, какие общие виды элементов (объекты какого класса) будут выполнять работу, требуемую от приложения.

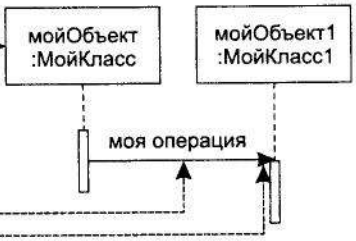
ОДИН ИЗ СПОСОБОВ ПОСТРОИТЬ ДИАГРАММУ ПОСЛЕДОВАТЕЛЬНОСТИ (1)

1. Определите вариант использования, диаграмму которого вы собираетесь строить.
2. Определите, какие объекты инициализируют вариант использования:
   - пользователь;
   - объект класса; в этом случае назовите класс, назовите объект.
3. Нарисуйте прямоугольник, изображающий этот объект в левом верхнем углу. Используйте UML-нотацию «объект: класс».
4. Нарисуйте продолговатый прямоугольник под ним, изображающий выполнение операции.
5. Нарисуйте стрелку, направленную вправо от него, для указания вызванной функции.



ОДИН ИЗ СПОСОБОВ ПОСТРОИТЬ ДИАГРАММУ ПОСЛЕДОВАТЕЛЬНОСТИ (2)

6. Определите, какой объект выполняет инициированную операцию:
   - объект класса: назовите класс, назовите объект.
7. Пометьте стрелки именем операции.
8. Покажите начало процесса с помощью вытянутого прямоугольника.
9. ...Продолжайте для каждого нового утверждения варианта использования.



4.5. Организация D-требований

4.5.1. Почему так важно организовать детальные требования

Чтобы понять значение аккуратно организованных D-требований, представьте себе следующую довольно беспорядочную попытку написания D-требований для игры *Встреча*. Обратите внимание, что эти требования все еще «сырые» и непроверенные.

- Каждый персонаж в видеоигре *Встреча* должен иметь имя.
- Каждый игровой персонаж имеет одинаковый набор характеристик, значение каждой представлено вещественным числом.

- В игре *Встреча* на вычисление результатов контакта должно уходить не более одной секунды.
- Каждая зона должна иметь конкретный набор «необходимых характеристик». Например, зоны сражений требуют силы и выносливости; зоны жилых комнат требуют внимательности и ума.
- Когда два персонажа игры оказываются в одной зоне одновременно, они могут по выбору или принудительно вступить в контакт друг с другом.
- У каждого игрового персонажа должно быть некоторое число очков-жизней.
- Сумма значений характеристик игрового персонажа, соответствующих требованиям зоны, называется значением персонажа в зоне. Во время контакта система должна сравнивать значения персонажей в зоне и вычислять результат контакта.
- Имя каждого персонажа может содержать не более 15 символов.

Неорганизованный список, подобный приведенному выше, быстро превращается в неуправляемый по мере разрастания.

- Такой список трудно понять в целом, даже пока он не вырос до сотен, если не тысяч элементов.
- Требования имеют разные типы: например, требования *к производительности* должны обрабатываться иначе, чем требования *поведения*.
- Некоторые требования можно естественным образом объединить с другими.
- Трудно найти какое-либо конкретное требование.

4.5.2. Способы организации детальных требований

D-требования можно организовать с помощью нескольких схем:

- по *основным свойствам* (предоставляемый вовне сервис, обычно определяется с помощью пар стимул—реакция). Этот способ организации часто воспринимают как «требования», имея в виду, что требования сгруппированы по различным свойствам программы. Заметьте, что само по себе это не предоставляет никакой систематической организации, поскольку позволяет переходить от свойства одной части программы к свойству абсолютно другой части программы;
- по *режиму* (например, системы управления радарами могут иметь *тренировочный*, *нормальный* и *аварийный* режимы);
- по *вариантам использования* (иногда еще называется *по сценариям*). Эта организация будет подробно рассмотрена позднее. Идея заключается в том, что большинство детальных требований являются частью варианта использования;
- по *классу*. Это объектно-ориентированный стиль, подробно объясненный далее. В этом способе организации мы классифицируем требования по классам. Такой способ организации использован в нашем примере;

- по *иерархии функций* (то есть путем разбиения программы на множество высокоуровневых функций и последующего разбиения их на подфункции и т. д.) Например, требования для программы домашнего бюджета можно разбить на (1) функции проверки, (2) функции сбережений и (3) функции инвестирования. Функции проверки могут затем быть разложены на функции чековой книжки, баланса счета, составление отчетов и т. д. Это традиционный способ упорядочивания детальных требований;
- по *состояниям* (то есть путем указания детальных требований, применимых к каждому состоянию). В частности, требования для программы, управляющей химическим процессом, лучше всего классифицировать по состояниям, в которых может находиться процесс (*начало*, *реакция*, *охлаждение* и т. д.). Внутри классификации каждого состояния перечислены события, влияющие на программу, находящуюся в конкретном состоянии.

Классификация по состояниям может быть уместна, если требования для каждого состояния сильно отличаются. Например, бухгалтерская система может вести себя по-разному в зависимости от ее состояний, таких как *Конфигурация*, *Исполнение* или *Сохранение*. Хотя требования примера игры *Встреча* можно организовать по состояниям, мы решили, что это было бы не так удобно, как их организация по классам.

Ниже приведены возможные способы организации D-требований:

- по свойствам;
- по вариантам использования;
- по классам;
- по иерархии функций;
- по состояниям.

Следующие способы классификации D-требований взяты из стандарта IEEE 830-1993. Пользователи подобных схем могут добавлять разделы по необходимости. Например, в объектно-ориентированной классификации не хватает раздела, эквивалентного разделу 3.4 в не объектно-ориентированной классификации «Требования логических баз данных». Пример в конце главы использует модифицированную форму объектно-ориентированного стиля IEEE и включает раздел для вариантов использования.

IEEE 830-1993: Детальные требования ООП и других стилей

3. Конкретные требования (не ОО-формат)

3.1. Внешние интерфейсные требования

3.2. Функции

3.3. Требования к производительности

3.4. Логические требования базы данных

3.5. Ограничения проектирования

3.5.1. Соответствие стандартам

3.6. Атрибуты программной системы
3.6.1. Надежность
3.6.2. Доступность
3.6.3. Защищенность
3.6.4. Поддержка
3.6.5. Переносимость
3.7. Организация конкретных требований
3.7.1. Системный режим
3.7.2. Пользовательский класс
3.7.3. Объекты
3.7.4. Свойства
3.7.5. Входной сигнал
3.7.6. Ответный сигнал
3.7.7. Функциональная иерархия
3.8.8. Дополнительные комментарии

3. Конкретные требования (ОО-формат)

3.1. Требования к внешнему интерфейсу
3.1.1. Пользовательские интерфейсы
3.1.2. Аппаратные интерфейсы
3.1.3. Программные интерфейсы
3.1.4. Коммуникационные интерфейсы
3.2. Классы/Объекты
3.2.1. Класс/Объект 1
3.2.1.1. Атрибуты (прямые или унаследованные)
3.2.1.1.1. Атрибут 1...
3.2.1.2. Функции (службы, методы, прямые или унаследованные)
3.2.1.2.1. Функциональное требование...
3.3. Требования к производительности
3.4. Ограничения проектирования
3.5. Атрибуты программной системы
3.6. Дополнительные требования

Можно посоветовать организовывать детальные требования в комбинацию классификаций. Например, в состояниях *Конфигурация*, *Исполнение* или *Сохранение* бухгалтерской программы можно было бы использовать организацию по характерным свойствам. Требования для автоматизированной системы завода можно было бы организовать на самом высоком уровне по функциям (*потребление*,

производство и *сборка*), а затем упорядочить по классам в каждой из получившихся функций.

Способ организации D-требований часто связан с возможной архитектурой программного приложения. Например, если проект будет объектно-ориентированным, должна использоваться организация по *вариантам использования* или по *классам*, поскольку это облегчит прослеживаемость. Это будет объяснено в следующих двух разделах. Если имеются действующие лица, осуществляющие все существующие и возможные требования отдельно, можно отдать предпочтение организации по *действующим лицам*.

4.5.3. Организация детальных требований по вариантам использования

Унифицированный процесс разработки программного обеспечения (USDP) использует то наблюдение, что многие требования естественно встречаются в последовательностях операций. Например, требование того, чтобы программа видеомагазина позволяла вводить название нового фильма, имеет место как часть последовательности сделки. Это варианты использования, которые еще часто называют *сценариями* (в UML *сценарием* часто называют экземпляр варианта использования). Набор вариантов использования для программы видеомагазина проиллюстрирован на рис. 4.10.

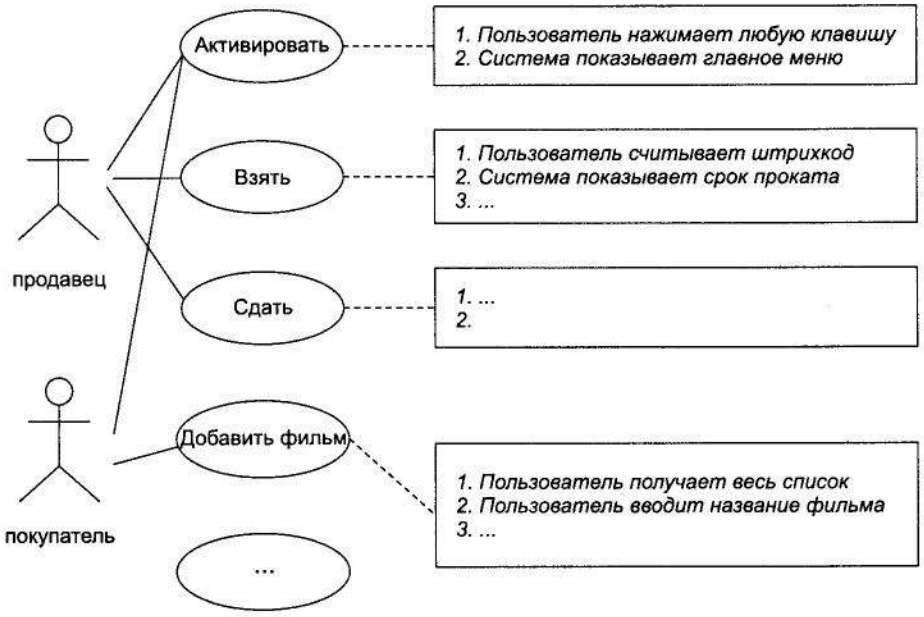


Рис. 4.10. Организация требований по вариантам использования: пример видеомагазина

Унифицированный процесс разработки программного обеспечения благоприятствует организации требований по вариантам использования. Если мы органи-

зуем D-требования таким образом, имеет смысл разрабатывать более крупные варианты использования из маленьких. Способ создания таких вариантов использования с помощью отношения *обобщения* UML показан на рис. 4.11. Вариант использования *B* обобщает вариант использования *D*, если *D* содержит все шаги *B* (обычно также и дополнительные шаги).

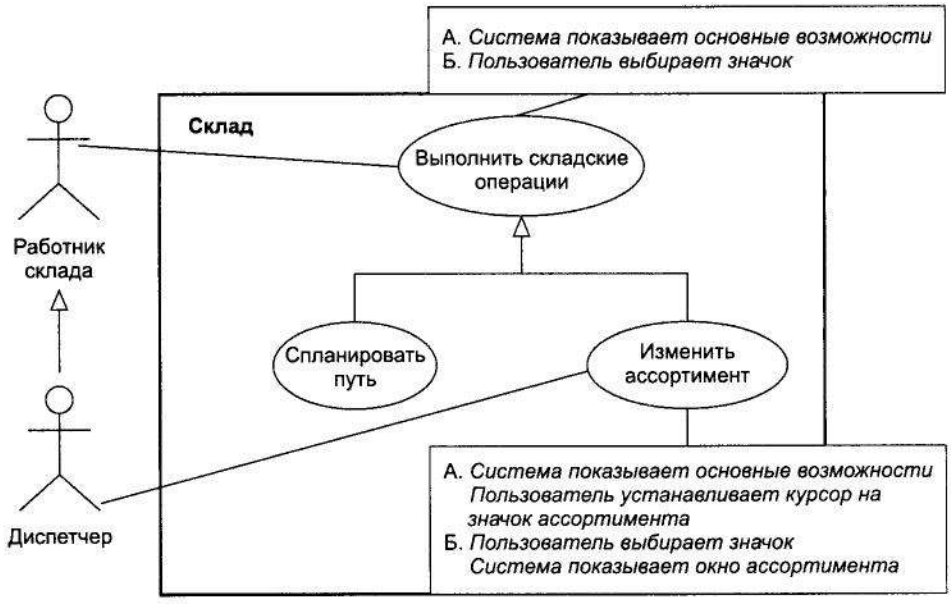


Рис. 4.11. Расширения и обобщения варианта использования

Для дополнительных способов получения новых вариантов использования из существующих см. [95].

4.5.4. Организация требований по классам

Мы сконцентрируемся на объектно-ориентированном стиле организации требований, в котором прежде всего надо определить классификацию — эквивалент выбора классов. Затем отдельные требования распределяют по получившимся категориям или классам. Для этого существует два подхода. Во-первых, можно рассматривать классы как способ организации требований, но не считать их обязательно используемыми в реальном проектировании. Во-вторых, мы можем использовать классы, разработанные для требований, в реальном объектно-ориентированном проектировании (и реализации), как мы и поступаем в нашем примере. Последний подход поддерживает взаимнооднозначную прослеживаемость между D-требованиями и методами, создавая соответствие между каждым функциональным D-требованием и функцией на целевом языке.

Один из недостатков этого подхода заключается в риске, связанном с тем, что позднее мы изменим используемые классы, нарушив соответствие между требо-

ваниями и классами [69]. Другой недостаток этой классификации в том, что здесь от нас требуется выбрать классы довольно рано в процессе разработки, причем многие считают, что мы таким образом эффективно выполняем проектирование. Рассмотрим в качестве примера игру *Встреча*. Выбор таких классов, как *ПерсонажИгрока* и *Зона*, на стадии требований не причинит вреда, поскольку эти классы скорее всего будут согласовываться с любой выбранной архитектурой. Иными словами, эти классы скорее всего и будут использованы в реализации. С другой стороны, поскольку объекты *СоединениеЗоны* ссылаются на объекты соответствующих *Зон*, это может быть рассмотрено как решение проектирования.

Большим преимуществом организации требований по классам, которые будут использованы в проектировании, является поддержка жесткого соответствия между требованиями, проектом и реализацией. Это главное достоинство использования объектно-ориентированного принципа. Вдобавок для классов, соответствующих понятиям реального мира, гораздо больше вероятность повторного использования. Для многих программ выгода от использования метода организации требований по классам перевешивает все недостатки.

Типичная последовательность получения функциональных D-требований с использованием объектно-ориентированного стиля приведена ниже.

1. Процесс начинается с перечисления классов, упомянутых в вариантах использования.
2. Полученный набор классов обычно неполон, и следует попытаться найти остальные классы предметной области. Этот процесс описан ниже.
3. Для каждого из полученных классов выписывается вся необходимая функциональность программы, за которую отвечает данный класс. Это показано в примере в конце главы. Документ создается в форме атрибутов и функций. Например, «у каждого клиента будет имя» (атрибут класса *Клиент*) и «программа должна иметь возможность вычислять капитал каждого клиента» (функция класса *Клиент*).

   Каждый известный обязательный объект класса должен быть отдельно указан как требование к классу. Например, «*Концерн семьи Рокфеллер* должен быть клиентом».

   События, которые должны обрабатывать объекты класса, должны быть также определены.

   D-требования проверяются по ходу процесса.

   В идеале в это же время должны быть разработаны и планы тестирования для каждого D-требования, как будет объяснено далее.
4. D-требования затем проверяются и сравниваются с C-требованиями.
5. D-требования проверяются заказчиком, и лишь затем публикуются. Вспомните, что основной аудиторией для D-требований являются разработчики. Однако заказчики также существенно заинтересованы в деталях.

Итог этих шагов подводится на рис. 4.12.

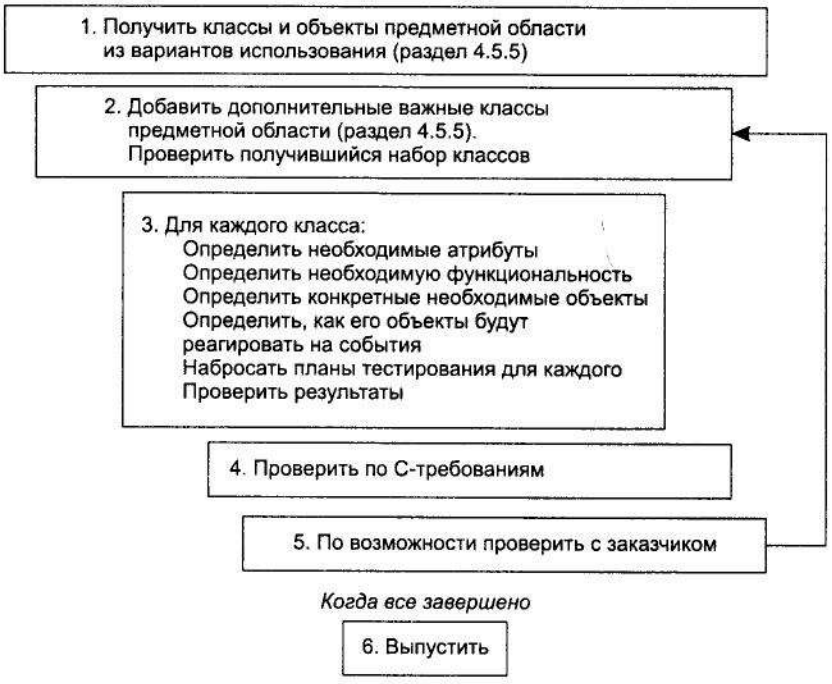


Рис. 4.12. Схема D-требований с использованием объектно-ориентированного стиля

Частой ошибкой, возникающей при организации требований по классам, является рассмотрение процесса определения D-требований как процесса проектирования. Для описания детальных требований должен использоваться простой язык. Например, следующий язык **приемлем**:

Должна существовать возможность получить число неоплаченных дней на любом счету.

Нижеследующая фраза **неприемлема**:

getDelinquentDays() возвращает количество неоплаченных дней на счету.

Другими словами, объектно-ориентированный подход используется только как принцип организации требований. Использование его для проектирования и реализации выполняется позднее.

4.5.5. Определение классов

Классы, которые будут использоваться в качестве нашего организующего принципа, должны определяться аккуратно и надежно. Это достигается путем определения классов предметной области, принадлежащих программе. Например, среди классов предметной области банковской программы могут быть классы *КлиентБанка* и *Кассир*, но не *Файл* или *БазаДанных* и даже не *Клиент* или *Операция*. Последние два класса не являются конкретными для нашего приложения. Использование классов предметной области является способом организации, обдумыва-

ния и отслеживания требований. Наша цель заключается в определении минимального, но достаточного набора классов предметной области, включающих все детальные требования.

В качестве другого примера набора классов предметной области представим программу, управляющую посещениями веб-сайта. Кандидатами в классы будут *ПосетительСайта*, *ПосещениеСайта* и *ЦельПосещения*. Требования, относящиеся к посетителю (например, данные о посетителях, показ профиля посетителя), будут собираться с помощью класса *ПосетительСайта*. Если программе требуется, чтобы мы отслеживали причины каждого посещения, класс *ЦельПосещения* также будет уместен. Соответствующие требования будут собраны внутри класса *ЦельПосещения*. Например, в требования к программе будет входить требование, согласно которому посетители должны заполнять форму с указанием их целей при посещении сайта.

Варианты использования являются основным источником классов предметной области. Вспомните диаграмму последовательности для примера игры *Встреча* (см. рис. 4.8).

За исключением случая, когда программа представляет собой преимущественно графический пользовательский интерфейс, автор советует по возможности отложить рассмотрение классов графического пользовательского интерфейса до стадии проектирования. Причина состоит в том, что это не основные классы предметной области и разработчики часто их изменяют. Для первых версий можно попробовать использовать интерфейс командной строки. С другой стороны, классы графического пользовательского интерфейса могут потребоваться рано, поскольку клиент просто хочет увидеть какую-то графику на ранних этапах. В примере нам пришлось рано добавить класс *ОкноХарактеристикИгрока*, поскольку он упоминается в варианте использования С-требования с тем же именем. Вспомните, что диаграмма последовательности использует следующие классы:

- *ИграВстреча*

  класс, имеющий один объект;
- *персонажИгрока*

  с объектом *главныйПерсонажИгрока*;
- *зона*

  с объектом *Гардероб*;
- *окноХарактеристикИгрока*

  класс графического интерфейса пользователя, добавленный для полноты варианта использования.

Классы *ГиперссылкаСоединения*, *СоединениеЗоны*, *Зона* и *ПерсонажИгрока* обязательны для диаграммы последовательности «Перейти в соседнюю зону» (рис. 4.13), которая соответствует одноименному варианту использования в С-требовании. Классы *ВнешнийПерсонаж*, *Контакт* и *ИзображениеКонтакта* вызываются в диаграмме последовательности «Вступить в контакт с внешним персонажем» (рис. 4.14). Нумерация в диаграмме последовательности показывает создание, а затем исполнение события *Контакт* для обработки результатов контакта. Это событие изменяет значения участников (шаг 2.1) в соответствии

с правилами контакта. Затем контакт вызывает свою операцию отображения результатов (шаг 3.1) и т. д.

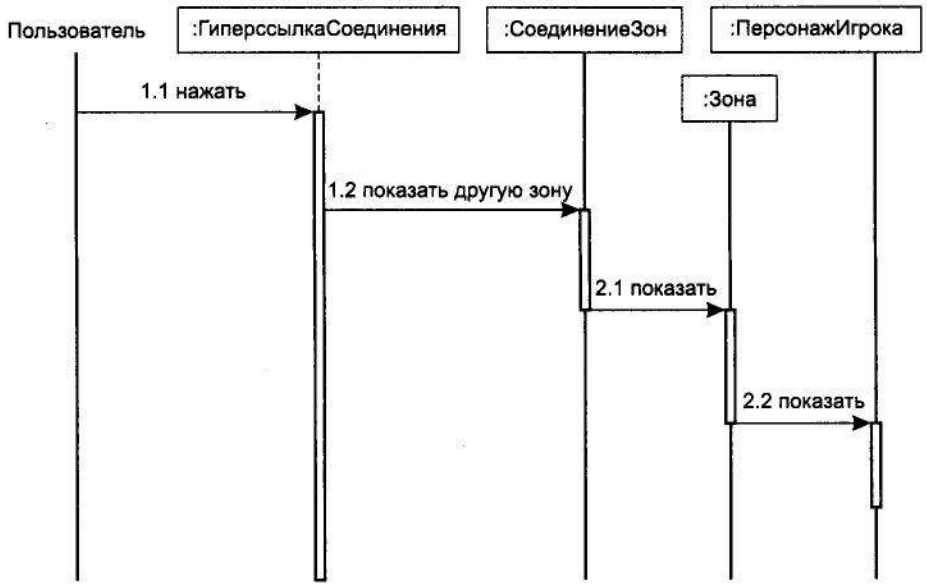


Рис. 4.13. Диаграмма последовательности для варианта использования «Перейти в соседнюю зону»

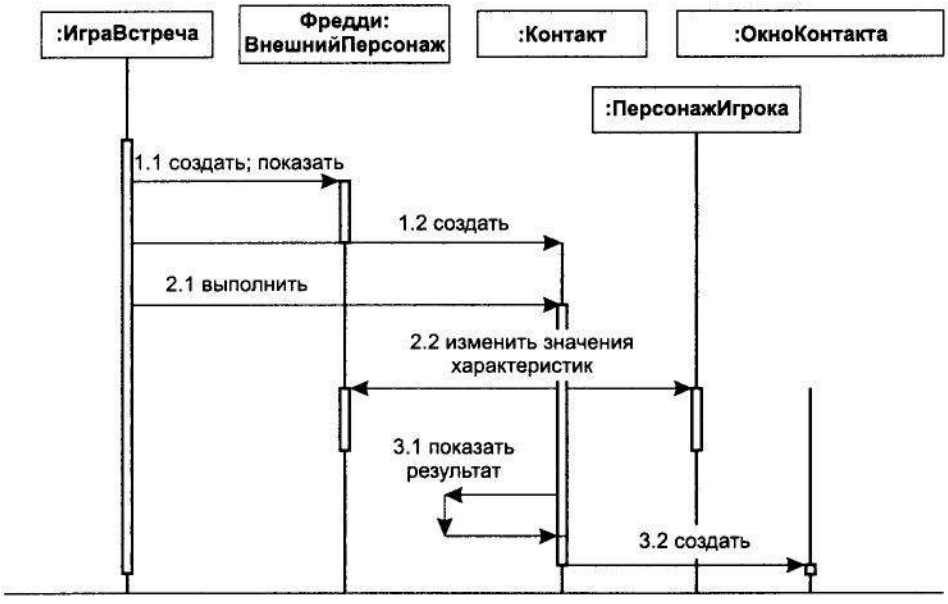


Рис. 4.14. Диаграмма последовательности для варианта использования «Вступить в контакт с внешним персонажем»

После получения классов из вариантов использования эффективным способом завершения определения ключевых классов предметной области является использование процесса «перечислить и урезать». Он состоит, во-первых, из перечисления всех возможных классов-кандидатов, которые вы только можете придумать, и во-вторых, из агрессивного удаления лишних кандидатов, после которого должны остаться лишь несколько самых важных.

1. Классы-кандидаты для игры *Встреча*, отобранные из диаграмм последовательности, вместе с существительными из предшествующего описания игры в этой главе показаны на рис. 4.15. UML-нотация для класса — это прямоугольник с именем класса.
2. Затем мы их фильтруем. Сначала обратите внимание, что гораздо проще добавить класс в дальнейшем, чем удалить класс, встроенный в проектирование и реализацию, так что если вы сомневаетесь относительно полезности класса-кандидата, смело удаляйте его.

Логика, использованная для конечного выбора классов предметной области в примере, приводится далее.

- *Встреча*: Изменить на *ИграВстреча*, чтобы подчеркнуть цель (возможно, нам еще понадобится и понятие «встреча»).
- *Игра*: не класс предметной области — слишком общий (возможно, мы повторно введем его, когда будем искать подходящие обобщения).
- *ПерсонажИгры*: слишком общий, чтобы быть классом предметной области (возможно, мы повторно введем его, когда будем искать подходящие обобщения).
- *Игрок*: более предпочтительно имя *ПерсонажИгрока* (более конкретно для класса).
- *ВнешнийПерсонаж*: OK (внешние персонажи действуют иначе, чем персонаж игрока).
- *ПерсонажВстречи*: OK (обобщение *ПерсонажаИгрока*, *ВнешнегоПерсонажа* и т. д. — все еще внутри предметной области программы).
- *Характеристика*: пропустить — попытайтесь обработать как простой атрибут класса *ПерсонажВстречи*.
- *Комната*: пропустить — не уверены, нужно ли это; уже есть *Зона*.
- *Дверь*: пропустить — не уверены, понадобится ли это.
- *Выход*: не уверены, понадобится ли: ведет в соседнюю зону — попробуйте как простой атрибут *Зоны*. Пока пропустить.
- *Правило*: пропустить — не уверены, понадобится ли это.
- *Зона*: OK [внимательный читатель заметит, что это дефектное решение].
- *Контакт*: OK.
- *Проход*: Нам нужно соединять зоны, но мы еще не знаем, в какой форме будут реализованы эти соединения. Вместо этого используйте *СоединениеЗоныВстречи*.

- *Результат*: пропустить — не ясно.
- *Сражение*: пропустить — не уверены, нужно ли это: уже есть *Контакт*.
- *Очки*: пропустить — попробуйте как атрибут других классов.
- *ОкноХарактеристикИгрока*: необходимо для выражения варианта использования «Инициализировать».
- *ОкноВыбораВыхода*: пропустить — не нужно (достаточно щелкнуть на гиперссылке выхода).
- *Карта*: пропустить — не обязательно на этом этапе: возможно, появится в будущих версиях.
- *ОкноКонтакта*: ОК — нужно для варианта использования, хотя мы постараемся отложить это, заменив все интерфейсом командной строки.

Рис. 4.15. Классы-кандидаты для игры Встреча

Получившиеся классы показаны на рис. 4.16. На рисунке также показаны наследственные связи между этими классами, отмеченные стрелкой с треугольником.

Существуют и другие способы наследования, в которых классы на рис. 4.16 потенциально связаны. Например, *СоединениеЗоныВстречи*, скорее всего, будет агрегировать два объекта класса *Зона*. Здесь наше опасение касается лишь основных классов программы, а также использования их для организации требований. Связи между классами показаны, где это необходимо. Использование наследования обеспечивает определенную свободу. Например, после утверждения требований для класса *ПерсонажВстречи* нам не нужно повторять требования для описания классов *ПерсонажИгрока* и *ВнешнийПерсонаж*. Класс *ПерсонажВстречи* показан курсивом на рис. 4.16, поскольку он абстрактный: это означает, что не будет других персонажей, помимо персонажей, управляемых игроком, или внешних.

Рис. 4.16. Классы для видеоигры Встреча с отмеченным наследованием

Описанный выше метод идентификации классов для классификации детальных требований показан в следующей врезке.

ОДИН ИЗ СПОСОБОВ ВЫБРАТЬ КЛАССЫ ПРЕДМЕТНОЙ ОБЛАСТИ ДЛЯ КЛАССИФИКАЦИИ ТРЕБОВАНИЙ

1. Разработать полный набор непересекающихся вариантов использования.
2. Создать диаграмму последовательности для каждого варианта использования. Не забыть отождествить классы и объекты.
3. Собрать классы, использовавшиеся в диаграммах последовательности.
4. Определить важные дополнительные классы.
5. Распределить подробные функциональные требования по этим классам.
   1) Указать каждый атрибут как отдельное требование.
   2) Указать каждый конкретный объект этого класса, который должен существовать.
   3) Указать каждую функцию, необходимую для объектов в этой классификации.
   4) Перечислить события, на которые должны реагировать все объекты этого класса.

Стандарт IEEE 830-1993 SRS предписывает обозначать место, в котором сформулирована цель каждого класса, а также ключевые атрибуты и функции. Стандарт призывает использовать десятичную систему нумерации для каждого класса (например, 3.2.5), требований атрибутов (например, 3.2.5.1) и каждого функционального требования (например, 3.2.5.2). Вместо этого мы будем организовывать классы по алфавиту, чтобы упростить задачи добавления и удаления классов. Такие изменения будут необходимы по ходу роста программы. Кроме того, важно нумеровать требования, чтобы иметь возможность управлять ими, так что мы сделаем это внутри каждого класса, как показано в примере.

Иногда нумеруют только *важные* требования, поскольку на этот момент следует отслеживать только их. Этот стиль организации детальных требований проиллюстрирован в [69]. Для облегчения отслеживания D-требований может помочь нумерация каждого из них, как это сделано в примере. Например:

3.2.A.7. Предпочтительные характеристики

[важно] У каждой зоны будет предпочтительный набор характеристик.

Добавление *желательных* и *необязательных* детальных требований выгодно по нескольким причинам. Во-первых, границы программы можно контролиро-

вать путем реализации требований в запланированном порядке. Во-вторых, формулировка будущих требований направляет разработчиков, помогая им принимать проектные решения, которые смогут удовлетворить будущие потребности.

Одной из техник указания требований, для которых уже проведены проектирование и реализация, является добавление статуса к каждому из требований. Например:

3.2.A.7. Предпочтительные характеристики

[важно, еще не реализовано] Каждая зона будет представима в одном из трех разных стилей.

Когда требование будет реализовано, фразу «еще не реализовано» можно убрать.

4.5.6. Правильный выбор класса для данного требования

Возможно, наиболее сложным вопросом при организации D-требований по классам является решение, какому классу приписать данное требование. Вот простой пример.

Требование. Каждый персонаж Встречи будет иметь имя.

Это требование должно быть отнесено к классу *ПерсонажВстречи*. А вот более сложный пример.

Требование. Каждый раз, когда главный персонаж игрока вступает в зону, эта зона и все персонажи, находящиеся в ней, будут показаны на мониторе.

Очевидно, классами-кандидатами для этой функции будут *ПерсонажИгрока* и *Зона*. Требование, очевидно, вызывает обработчик событий. Согласно интерфейсу, показанному в главе 3, персонаж игрока вступает в зону после того, как игрок щелкает на гиперссылке зоны. Поэтому естественным объектом для обработки события входа в зону будет зона, в которую вошел игрок, поскольку она должна владеть информацией относительно находящихся в ней персонажей. Зона, в которую вошел игрок, может показать себя и персонажи, которые находятся в ней. Поэтому требование, сформулированное выше, логично отнести к классу *Зона*.

4.5.7. Классификация объектов

Для программ необходимо присутствие конкретных объектов или экземпляров (а не классов). Например, наша программа с видеоигрой требует существования объектов *двор* и *гардероб* класса *Зона*. Где мы должны сформулировать эти требования? Существует по крайней мере три варианта. В первом варианте мы должны внести эти требования в класс, создающий объекты. Согласно диаграмме последовательности, *ИграВстреча* создает *гардероб*. Преимущество этого способа заключается в том, что такое решение отражается в коде, поскольку функция создания в объекте *ИграВстреча* будет прямо ссылаться на объект *гардероб*. Недостаток такого подхода заключается в том, что решение относительно того, какой объект будет участвовать в создании нашего объекта, с большой вероятностью изменится, и тогда требования также придется перемещать. Также следует отме-

тить, что другие объекты, хотя и не создают наш объект, также могут ссылаться на него чаще, чем создающий его объект.

Второй вариант заключается во введении дополнительного класса, агрегирующего конкретные объекты каждого важного класса (например, класс *Зоны*, агрегирующий объекты класса *Зона*). Это может оказаться неудобным, и в этом случае также могут быть добавлены классы, не являющиеся необходимыми. Рассмотрим третий способ: перечисление требований к объектам и классу, которому они принадлежат.Требования существования объектов *гардероб* и *двор* могут быть перечислены для класса *Зона* следующим образом:

Класс Зона

Все действия игры (в том числе и контакты) происходят в Зонах.

...

Требование Зоны (гардероб). Должен существовать объект класса Зона, названный «гардероб». Он будет выглядеть... Его предпочтительными характеристиками должны быть сосредоточенность и выносливость.

Требование Зоны (двор). Должен существовать объект класса Зона, названный «двор». Он будет выглядеть... Его предпочтительной характеристикой должна быть сила.

4.5.8. Связь с документацией тестов

По ходу классификации каждого D-требования следует выполнять некоторую работу по тестам для этого конкретного требования. Существует несколько преимуществ в написании тестов одновременно с требованием. Во-первых, такие действия помогают прояснить требование. Во-вторых, это переносит некоторую часть работы из фазы тестирования проекта в фазу требований. Это снимает некоторое давление на более поздних этапах проекта, когда временные рамки более ограничены.

Например, одно из требований выглядит так:

Требование NNN. Каждый игровой персонаж в видеоигре Встреча будет иметь уникальное имя, содержащее от 1 до 15 символов.

Требования с таким типом атрибута на самом деле определяются функциями *get-* и *set-*. Начало тестовых планов для этого требования показано в табл. 4.2. Подробно эти тесты описаны в главе 8.

Таблица 4.2. Входные тестовые данные и ожидаемый результат

Входные тестовые данные для требования NNN	Ожидаемый результат
Гарри	Гарри
X	X
« » (пусто)	« » (пусто)
123456789012345	123456789012345
1234567890123456	123456789012345
...	...

Напомним (см. главу 1), что параллельное тестирование является важной характеристикой инкрементальной разработки и *экстремального программирования*. На основе концепции параллельного тестирования была построена методология, называемая *методологией Y*. Она получила свое название в связи с тем, что ее диаграмма представляет собой водопад, начинающийся слева, с соответствующей последовательностью фаз тестирования справа, тем самым образуя «рукава» буквы «Y». Эти два потока сливаются вместе на этапе интеграции, после чего происходит тестирование системы (глава 9).

Детали

Эту часть главы можно изучать после прочтения последующих глав. Однако понимание материалов, изложенных в этой части, обязательно для разработки качественного программного продукта.

4.6. Качество детальных требований

Постоянно держа в голове потенциально возможные последствия неудач с формулированием деталей каждого требования (например, потерянный искусственный спутник), мы стараемся оценить качество как можно большего числа наших требований.

4.6.1. Роль контроля качества в анализе D-требований

Организация, контролирующая качество, просматривает D-требования. Например, в стандарте планирования контроля качества IEEE (730.1-1995) сказано, что план контроля качества программного продукта

«должен определять или ссылаться на стандартные практики, соглашения и метрики, которые в свою очередь должны использоваться на этапе определения требований. Должны быть указаны привлеченные стандарты..., которые должны быть выполнены в основных требованиях, и должна быть обеспечена их прослеживаемость. Формальные языки описания требований должны использоваться, где это возможно. Схемы, однозначно определяющие каждое требование, должны иметь основу...».

В стандарте подробно изложены форма и природа программных требований (раздел 3.6.2.1 IEEE 730.1-1995).

В больших проектах иногда следуют указаниям такого типа. Однако во многих средних и небольших организациях контроль качества вынесен в отдельный процесс, который начинает выполняться только после того, как все требования были определены. Иногда группу проверки качества даже просят по факту «про-

верить то, что вот это место было построено согласно спецификации». Типичной жалобой группы контроля качества является отсутствие адекватных требований, по которым можно было бы проверить работу программы. Иногда специалистам по контролю качества приходится создавать требования согласно самой программе и тому, как она была построена. Этот процесс называется *обратным проектированием* (глава 10). Однако ситуация меняется к лучшему по мере того, как организации-разработчики начинают относиться к процессу более серьезно.

4.6.2. Метрики для анализа D-требований

Выборочное использование метрик максимизирует отдачу от материальных вложений в проверку, фокусируя процесс проверки на важных, измеренных результатах.

Каждая метрика дает выгоду, но стоит денег и времени для сбора, хранения, анализа и составления отчета. Искусство использования метрик заключается в оптимизации отношения затрат к получаемой выгоде. Это зависит от традиций организации, состояния проекта, природы проекта и многих других факторов. Сомнительной практикой является сбор метрик просто потому, что они могут пригодиться в будущем. Автор наблюдал огромные количества собранных данных, покрытых толстым слоем пыли, и все только потому, что кто-то посчитал, что они могут пригодиться, хотя и не сформулировал, как. Процесс сортировки для отбраковки метрик является полезной практикой, классифицирующей их на «обязательные» метрики, те метрики, которые «хорошо было бы иметь» и «остальные». Несколько попыток использования «остальных» метрик прояснят их стоимость и прибыль.

Далее приведен список метрик контроля качества, куда включены метрики анализа требований из стандарта IEEE 982.2-1988.

Метрики контроля качества детальных требований включают в себя:

- метрики того, насколько хорошо написаны требования:
  - процент однозначных детальных требований (IEEE-метрика 6);
  - степень законченности (полнота) (IEEE-метрики 23 и 35);
  - процент неочевидных D-требований (в объектно-ориентированном стиле это измеряется процентом требований, размещенных в неправильном классе);
  - процент требований, которые:
    - не тестируются;
    - не прослеживаются (IEEE-метрика 7);
    - не отсортированы по приоритетам;
    - не элементарны (можно разбить на части);
    - не согласуются с остальными требованиями (IEEE-метрики 12 и 23);
- метрики эффективности проверки требований:
  - процент пропущенных или дефектных требований, найденных за каждый час проверки;

- метрики эффективности процесса анализа требований:
  - стоимость каждого D-требования:
    - общая (общее затраченное время/число D-требований);
    - критическая (стоимость получить еще одно);
    - скорость, с которой детальные требования могут быть
      - изменены;
      - удалены;
      - добавлены;
- метрики полноты требований:
  - можно оценить после официального завершения сбора D-требований исходя из скорости, с которой требования
    - изменяют;
    - добавляют.

Метрики полезны, когда их целевые значения определены заранее. Например, базируясь на опыте предыдущих проектов, мы скажем, что требования будут считаться законченными, когда скорость внесения изменений и добавлений будет меньше 1 % в неделю.

4.6.3. Инспектирование анализа D-требований

Читателю предлагается обратиться к главе 1 для общего определения процесса инспектирования.

Детальные требования (или D-требования) представляют собой первые документы процесса разработки, которые можно изучать, сравнивая с предшествующей документацией (C-требованиями). Инспекторы готовятся к проверке, перечитывают C-требования (например, IEEE 830-1993, разделы 1 и 2), и сравнивают с ними детальные требования.

4.6.3.1. Пример непроверенных D-требований

В этом разделе приведена версия D-требований, для которой мы выполним пример проверки, вводя результаты в таблицу (см. табл. 4.3). Окончательная версия этих требований, получившихся после проверки, показана в примере в конце главы.

Зона. Требование 1 (название *Зоны*). [Еще не проверено] У каждой зоны должно быть имя, содержащее от 1 до 15 символов.

Зона. Требование 2 (изображение *Зоны*). [Еще не проверено] Должен быть рисунок в формате GIF, показывающий объект *Зона*.

Зона. Требование 3 (метод *Показать*). [Еще не проверено] Каждый раз, когда персонаж игрока входит в зону, эта зона и все персонажи, находящиеся в ней, должны быть показаны.

Зона. Требование 4 (объект *двор*). [Еще не проверено] Должен быть объект класса *Зона* с именем *«двор»*. Его вид показан на рис. 4.38.

Зона. Требование 5 (объект *гардероб*). [Еще не проверено] Должен быть объект класса *Зона* с именем *«гардероб»* и пустым изображением на заднем плане. *Гардероб* будет соседней зоной для двора.

Встреча. Требование 1. (Вступить в контакт с внешним персонажем) [Еще не проверено] Каждый раз при контакте должны производиться следующие вычисления: сумма значений характеристик игрового персонажа по отношению к зоне будет называться *значением* в зоне [в этой версии все значения будем считать одинаковыми]. При контакте система сравнивает значения персонажей в зоне и передает сильнейшему половину значения слабого. Например, предположим, что игрок вступает в контакт с внешним игроком в зоне, требующей выносливости и внимательности, и *ps* — это значение выносливости игрока и т. д. Считая, что $p_s + p_a > f_s + f_a$, получим $p'_s = p_s + f_s / 2$, $p'_a = p_a + f_a / 2$, $f'_s = f_s / 2$, $f'_a = f_a / 2$, где штрихом отмечены новые значения.

ПерсонажВстречи. Требование 1 (Имя игрового персонажа). [Еще не проверено] Каждый игровой персонаж в игре *Встреча* будет иметь уникальное имя, содержащее от 1 до 15 символов.

ПерсонажВстречи. Требование 2 (Характеристики игровых персонажей). [Еще не проверено] У каждого игрового персонажа имеется одинаковый набор характеристик, значение каждой из которых представлено вещественным числом. Изначально все значения равны $100/n$, где n — число различных характеристик. Характеристиками являются *внимательность*, *сосредоточенность*, *ум*, *терпение* и *сила*.

ПерсонажВстречи. Требование 3 (Внешний вид игрового персонажа). [Еще не проверено] Каждый игровой персонаж будет показан с помощью рисунка, занимающего не более 1/8 части экрана монитора.

ПерсонажВстречи. Требование 4 (Контакт с внешним персонажем). [Еще не проверено] Каждый раз, когда персонаж вступает в зону, содержащую другой игровой персонаж, и один из персонажей управляется игроком, персонаж игрока может либо по желанию, либо принудительно вступить в контакт с другим персонажем. Есть ли выбор, будет определяться игрой случайным образом с вероятностью 50 %.

ИграВстреча. Требование 1 (Встретить игровой объект) [Еще не проверено] Будет существовать только один объект класса *ИграВстреча*.

ВнешнийПерсонаж. Требование 1 (Объект «Фредди», внешний персонаж). [Еще не проверено] Будет существовать внешний персонаж с именем *Фредди*, все значения характеристик которого равны и внешний вид которого показан на рис. 4.28.

ПерсонажИгрока. Требование 1 (Возможность настройки). [Еще не проверено] Когда в зоне нет внешних персонажей, игрок может устанавливать значения характеристик с помощью объекта *ОкноХарактеристикИгрока*, сохраняя сумму значений характеристик неизменной.

ПерсонажИгрока. Требование 2 (Главный персонаж игрока). [Еще не проверено] У игрока будет полный контроль над конкретным игровым персонажем, называемым *главным персонажем*.

ПерсонажИгрока. Требование 3 (Очки-жизни). [Еще не проверено] Игра будет рассчитывать сумму значений характеристик персонажа, которая будет называться *количеством очков-жизней*.

4.6.3.2. Пример результатов проверки D-требований

В этом разделе мы покажем типичные результаты проверки D-требований.

Можно сделать одно замечание относительно этого набора в целом: требования недостаточно поддерживают преобразование игры в конкурентоспособный продукт. Более конкретным дефектом является то, что требования не определяют должным образом задержку, связанную с установкой значений характеристик игрока. Во время этой задержки игрок может быть вовлечен в контакт в неподготовленном состоянии. (Если задержка слишком мала, игрок просто устанавливает необходимые в зоне характеристики по максимуму, и игра не представляет особого интереса.) Давайте исследуем требования из предложенного списка по отдельности.

Пример формы, которую можно использовать для проверки D-требований применительно к созданному выше списку, приведен в табл. 4.3. Указанные в табл. 4.3 свойства определены ранее в разделе 4.3 (согласно [92]). Большинство метрик, описанных в этом разделе, можно рассчитать согласно этой таблице.

Вот примечания, отмеченные в таблице как «Прим.»:

1. Может ли персонаж или зона иметь имя без символов вообще?
2. Число 15 жестко закреплено.
3. Только один?
4. Если игрок управляет несколькими персонажами, должны ли показываться все зоны, или это касается только главного персонажа?
5. Заполнение всего экрана монитора?
6. Должно быть проще добавлять новые характеристики или удалять их.
7. Когда появляется Фредди?
8. В будущих версиях персонажи могут видоизменяться.
9. Проясните, что остается без изменения.
10. Может ли значение характеристики быть отрицательным?
11. Двусмысленность, поскольку игрок не может управлять всем, что происходит с главным персонажем, постоянно.

Уточните термин «полное управление».

1. Разрешен ли абсолютно любой символ клавиатуры?
2. Проверить справедливость с заказчиком.
3. Неясно, насколько изменяемым это должно быть.
4. Трудно ответить «полный», поскольку это нечетко. См. ссылки на примечания в столбце «Ясность».
5. Мы предполагаем, что у заказчика есть некоторая дополнительная информация относительно того, как должен выглядеть двор.
6. А есть ли выходы из гардероба в другие зоны?
7. Нечетко написано: может привести к неправильному пониманию.

Таблица 4.3. Пример проверки результатов D-требований

Требование	Прослежи-ваемость назад	Полнота	Согла-сован-ность	Выполни-мость	Одно-значность	Ясность	Точность	Модифи-цируе-мость	Тестируе-мость	Прослежи-ваемость вперед
Зона Требование 1	Прим. 14	Прим. 13	Да	Да	Прим. 1	Да	Прим. 1	Прим. 2	Прим. 1, 2	Да
Зона Требование 2	Да	Да	Да	Да	Прим. 3	Да	Прим. 3	Прим. 15	Да	Да
Зона Требование 5	Да	Прим. 17	Прим. 17	Да	Прим. 3	Прим. 3	Прим. 5	Да	Да	Да
Зона Требование 3	Да	Да	Да	Да	Да	Да	Прим. 18	Прим. 15	Да	Да
Зона Требование 4	Да	Прим. 19	Да	Да	Да	Да	Да	Да	Да	Да
Контакт Требование 1	Прим. 14	Да	Да	Да	Да	Да	Да	Прим. 15	Да	Да
ПерсонажВстречи Требование 1	Да	Прим. 13	Да	Да	Прим. 1	Да	Прим. 1	Прим. 2	Прим. 1, 2	Да
ПерсонажВстречи Требование 2	Да	Да	Да	Да	Да	Прим. 20	Да	Прим. 6	Да	Прим. 21
ПерсонажВстречи Требование 3	Да	Да	Да	Да	Да	Да	Да	Прим. 15	Да	Да
ПерсонажВстречи Требование 4	Прим. 23	Да	Да	Да	Да	Да	Да	Прим. 7	Да	Да
ИграВстреча Требование 1	Да	Да	Да	Да	Да	Да	Да	Прим. 24	Да	Д
ВнешнийПерсонаж Требование 1	Прим. 25	Да	Да	Да	Да	Да	Прим. 8, 9	Да	Да	Да
ПерсонажИгрока Требование 1	Да	Да	Да	Да	Да	Прим. 10	Да	Да	Да	Да
ПерсонажИгрока Требование 2	Да	Да	Да	Да	Прим. 11	Прим. 12	Прим. 12	Прим. 15	Прим. 12	Да
ПерсонажИгрока Требование 3	Да	Да	Да	Да	Прим. 22	Прим. 22	Да	Да	Да	Да

8. Обычно предпочтительнее, чтобы каждому атрибуту соответствовало отдельное требование. Это не обязательно, поскольку разные характеристики можно обрабатывать одинаково.
9. Рассчитывать в любое время? По требованию? Показывать всегда?
10. Подробности не упомянуты в С-требованиях: проверить с заказчиком.
11. Если можно, объясните термин «с вероятностью 50 %».
12. Для интернет-версий может быть необходимо иметь более одного объекта *ИграВстреча*. Мы не будем исключать такую возможность на будущих итерациях.
13. Неясно, как это можно изменить.
14. Написано ли требование с учетом того, что его можно будет проследить в коде, его реализующем?

Вдобавок к этим пунктам IEEE определяет метрику полноты. Как описано в стандарте 982.2-19888 A35.1, это формула, затрагивающая 18 наблюдаемых величин (например, «число параметров условия без обработки») и 10 весовых коэффициентов (например, относительная важность «использования определенной функции»). Попросту говоря, она вычисляет степень наличия «дыр» в наборе D-требований.

4.7. Использование инструментов для анализа требований

Инструментальные средства могут упростить процесс формирования требований и управления ими, например с помощью сортировки, расстановки приоритетов, назначения и прослеживания. Одной из выгод от использования инструментальных средств является информация о том, кто и когда работал над каким требованием. Инструментальные средства также помогают управлять «возникновением лишних свойств» — процессом появления в программе свойств, не являющихся жизненно важными. С помощью подходящих инструментов руководитель проекта может с большей легкостью определить статус анализа требований. Он может определить, например, какая часть важных D-требований была реализована и полностью протестирована группой контроля качества.

В простых проектах большинство этих операций можно выполнить, используя простую электронную таблицу, доступную через Интернет (табл. 4.4). Надпись «Спроектировано для» указывает, что требование учтено в проекте. «Модуль протестирован» означает, что код, реализующий данное требование, прошел модульное тестирование. «Сборка протестирована» означает, что программа была протестирована на предмет реализации требования.

Таблица хранится как часть документа статуса проекта. Ячейки этой таблицы можно связать посредством гиперссылок с соответствующими частями документов проекта.

Таблица 4.4. Пример таблицы для учета требований

Требование №	Приоритет			Статус							
					Степень готовности		Готово для инспектирования	Спроектировано для	Модуль протестирован	Сборка протестирована
	Важно	Желательно	Необязательно	Не начато	1/3	2/3	Проинспектировано			
Ответственный разработчик										
...	...	...	...	...	...	...	...	...	...	...

Гиперссылки можно использовать для поддержки централизованной разработки D-требований (например, чтобы застраховаться от повторений). Так, гиперссылки из исходного кода на соответствующее D-требование можно создать с помощью таких инструментов, как *Javadoc*. Javadoc конвертирует некоторые комментарии к Java-коду в HTML-документ, описывающий классы и их методы (см., в частности, [102]). Благодаря вставке гиперссылок на SRS внутри таких комментариев, HTML-документ, созданный с помощью Javadoc, также имеет гиперссылки на SRS. Это иллюстрирует представленный ниже пример, где на детальное требование, соответствующее методу `EngagingForeignCharacter()` (Вступить в контакт с внешним персонажем), идет ссылка из документа, сгенерированного Javadoc из исходного кода:

```
/**
<a href="RequAnal\# EngagingForeignCharacter">
        Engagement Requirement 1
        («Вступить в контакт с внешним персонажем»)
</a>
…Комментарии по реализации…
*/
    Цель этого метода дана в SRS.
    Цель не повторяется в исходном коде.
public engageForeignCharacter ( ... )
{
    ...
}
```

4.8. Формальные методы для спецификации требований

4.8.1. Введение в формальные спецификации

Математика хороша для выражения статического состояния, другими словами, для ответа на вопрос «что?». Это отличается от вопросов типа «как?», на которые отвечают процедуры и алгоритмы. Поскольку спецификации требований

в основном описывают состояние программы до и после действий, математическая нотация может быть более подходящей, чем естественный язык, для определения детальных требований. Использование математики в этом контексте является частью так называемых *формальных методов*. Формальные методы удобны для обученных математике инженеров. Этот раздел имеет своей целью познакомить читателя с идеей формальных методов для формулирования требований. Полное описание формальных методов спецификации вы найдете в [42] и [36].

Многие ученые верят, что математика играет важную роль в определении всех важных деталей, которые так легко могут испортить самый продуманный проект. Они считают, что такое использование математики может предотвратить пустую трату денег, разрушение имущества и даже человеческие потери (например, в жизненно важных программах).

Как пример рассмотрим следующее определение процедуры:

Возвращает отсортированный массив, состоящий из элементов массива A.

Это простое на первый взгляд требование поразительно наполнено неопределенностями. Элементы A могут иметь несколько ключей сортировки. Даже если A — это массив целых чисел, термин «сортировка» должен быть определен: отсортировать по четным и нечетным? по убыванию? Даже если порядок подразумевается возрастающим, требование остается неоднозначным.

Например, если $A = (4, 6, 3, 4, 6, 8)$, все три приведенные ниже массива удовлетворяют нашему требованию:

(3, 4, 6, 8), (3, 4, 4, 6, 8) и (3, 4, 4, 6, 6, 8).

Работа с требованием, выраженным в естественной форме, может упростить задачу. Например:

Возвращает отсортированный элемент всех отдельных элементов массива A.

К сожалению, слово *отдельный* также неоднозначно. Так можно до бесконечности перефразировать требование, но это будет лишь борьбой с математикой. Мы вернемся далее в главе к этому конкретному примеру.

Общая нотация для формального выражения требований называется *Z-спецификацией*. Z-спецификации являются стандартным способом описания требуемого состояния до и после процедуры. Ниже приведена некоторая выборка из Z-нотации.

4.8.1.1. Математическая нотация

$\Rightarrow$ означает *импликацию* или *логическое следование*. Таким образом, $P \Rightarrow Q$ означает: «если утверждение P истинно, то утверждение Q также истинно».

$\forall$ означает *для любого*. Например, $\forall$ живого человека L, L имеет голову.

$\wedge$ означает «и»; $\vee$ означает «или».

N означает множество натуральных чисел.

Если A множество, то $a \in A$ означает, что элемент a принадлежит множеству A. Например, $17 \in N$.

$A \subseteq B$ означает, что A является *подмножеством* B, то есть каждый элемент множества A является элементом множества B. Например, $E \subseteq N$, где E — множество четных чисел.

$A \times B$, где A и B множества, означает *прямое произведение* множеств, то есть множество пар элементов (a, b), где $a \in A$, $b \in B$.

Конечные множества можно обозначать с помощью фигурных скобок и запятых. Например, {4,7,2} означает множество, состоящее из элементов 4, 7 и 2. В этой нотации элементы не повторяются и их порядок не имеет значения.

$\{x: P\}$ означает множество элементов x, обладающих свойством P. Например, $\{x: x^2 = x + 3\}$ означает множество чисел, квадраты которых равны самому числу плюс 3.

Если S конечное множество, то card(S) означает количество элементов S. Например, card($\{g, e, q\}$) = 3.

4.8.1.1.1. Функции

Функция f из множества A в множество B — это подмножество прямого произведения $A \times B$, причем ни один элемент из A не встречается более одного раза в качестве первого элемента пары. Это обозначается так: $f: A \to B$. В формальной записи: $f \subseteq A \times B \wedge [[(a, b_1) \in f \wedge (a, b_2) \in f] \Rightarrow [b_1 = b_2]]$.

Областью определения функции называется множество элементов, которые встречаются в качестве первых элементов множества пар, образующих функцию. Например, областью определения функции $g: N \to N$, где $g = \{(3, 5), (7, 1), (8, 5)\}$, является множество {3, 7, 8}.

Областью значений функции называется множество элементов, которые встречаются в качестве вторых элементов множества пар, образующих функцию. Например, областью определения функции $g: N \to N$, где $g = \{(3, 5), (7, 1), (8, 5)\}$, является множество {5, 1}.

Если элемент y принадлежит области значений функции f, то $f^{-1}(y)$ *означает множество элементов из области определения, которые отображаются на элемент* y. *Формально* $f^{-1}(y) = \{x: f(x) = y\}$. *Например, для функции g, определенной выше,* $g^{-1}(5) = \{3, 8\}$.

$f : R \mapsto P$ означает, что f является *частичной* функцией из R в P, то есть, где $Q \subseteq R$. Например, $R = \{4, 7, 2\}$, $P = \{8, 1, 5\}$, $Q = \{4, 7\}$, $f(4) = 5$, $f(7) = 8$.

$a \mapsto b$ означает, что a отображается в b в контексте определяемой функции. Например, для определенной выше функции f $4 \mapsto 5$ и $7 \mapsto 8$.

Пусть $f : A \mapsto B$ и $g : A \mapsto B$. Тогда $f \oplus g$ означает *расширение* (или *доопределение*) функции g функцией f. Если x принадлежит области определения g, то $\phi \oplus g(x) = g(x)$, а если x не принадлежит области определения g, но принадлежит области определения f, то $f \oplus g = f(x)$. Так, используя функцию f из предыдущего примера и определяя $g(4) = 11$ и $g(8) = 3$, имеем: $f \oplus g(4) = 11$, $f \oplus g(8) = 3$ и $f \oplus g(7) = 8$.

4.8.2. Примеры формальных спецификаций

Z-спецификация состоит из двух соединенных прямоугольников с отсутствующими углами. Верхний прямоугольник описывает типы входных данных, параметров

и выходных данных. Нижний прямоугольник описывает состояние (ситуацию) после применения процедуры. Штрих после имени переменной указывает на значение переменной после выполнения процедуры. Например, x' — это значение переменной x после выполнения определенной процедуры. Входные данные обозначаются вопросительными знаками. (Вместо слов «на вход подается z» мы просто пишем $z?$.) Как только вы ближе познакомитесь с этой нотацией, вам станет удобнее ее использовать.

Этот раздел показывает, как использовать Z-спецификации, во-первых, для определения требований к приращению таблицы и, во-вторых, для выполнения поиска в таблице.

Пример 1: Приращение таблицы

Предположим, что мы хотим точно определить процедуру, которая получает на вход два целочисленных значения l и r и обновляет двумерную таблицу t, содержащую целые числа, путем либо добавления пары (l, r), если l в таблице отсутствует как первый элемент, либо изменения значения r, если l присутствует в таблице как первый элемент. Предполагается, что в таблице t нет повторяющихся первых элементов: на самом деле такие таблицы являются функциями из целых в целые числа.

Мы построим Z-спецификацию приращения таблицы целых чисел без повторяющихся первых элементов. В ней сказано, что t обновляется процедурой *Приращение* в новую функцию t'. Функция t' ведет себя аналогично t за исключением отображения l в r.

Например, пусть t — это следующая таблица:

5	7
2	11
1	2

Вызов функции *Приращение* с входной парой (2, 4) вернет таблицу:

5	7
2	4
1	2

а вызов функции *Приращение* с входной парой (3, 6) вернет таблицу:

5	7
2	4
1	2
3	6

Z-спецификация изображена на рис. 4.17 и 4.18. Для читателя, достаточно хорошо знакомого с Z-нотацией, эта спецификация предельно однозначна.

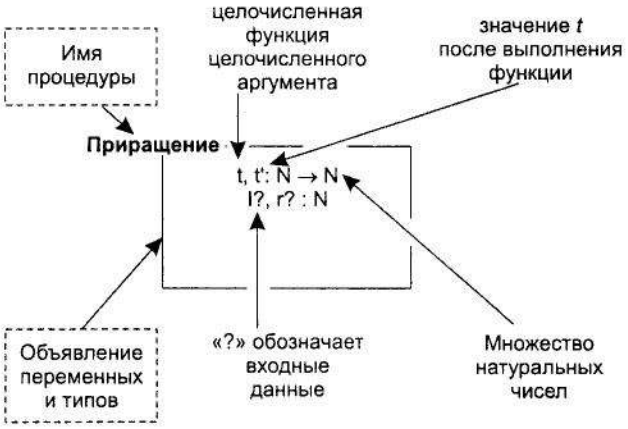


Рис. 4.17. Частичная Z-спецификация для Приращения таблицы: объяснение символов

Рис. 4.18. Полная Z-спецификация для Приращения таблицы

Пример 2: Поиск

На рис. 4.19 определена функция *Поиск*, которая просматривает все записи в таблице и возвращает результат (соответствующее целое число), если запись найдена, либо нуль в противном случае. Согласно Z-спецификации, t — это таблица, *I?* — входные данные (которые мы ищем) и *r!* — целочисленный результат, если *I?* найдена в левом столбце таблицы. Например, применение функции *Поиск* к таблицам, приведенным выше, со значением *I?*, равным 5, даст результат 7 для *r!*. Спецификация вернет нулевое значение *r!*, если в левом столбце таблицы *I?* не найдено.

Вот как можно прочитать результат (внизу).

1. *Либо* входной параметр *I?* не принадлежит области определения t, тогда вернуть нуль и оставить t без изменений.
2. *Либо* входной параметр *I?* принадлежит области определения t, тогда *r!* — это результат применения t к *I?*, t остается без изменений.

Почему нам приходится отдельно определять, что t остается без изменений? Потому что если мы не определим этого, мы не сможем пожаловаться на реализацию, меняющую t (возможно, для удобства программирования), поскольку такая реализация тоже будет удовлетворять требованиям.

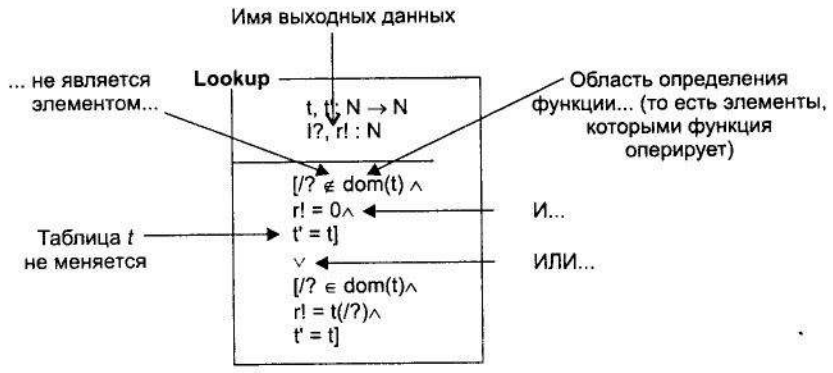


Рис. 4.19. Z-спецификация для Поиска записи в таблице

Пример 3: Сортировка

В этом примере мы вернемся к примеру сортировки, приведенному в начале этого раздела. Вспомните, что требование на (несовершенном) естественном языке было таким:

Вернуть отсортированный массив всех отдельных элементов массива.

В математике массивы — это функции с областью определения $\{1, 2, 3, ..., n\}$, где n — некоторое положительное целое число. Например, массив $A = (4, 6, 3, 4, 6, 8)$ эквивалентен функции, показанной в табл. 4.5.

Таблица 4.5. Массив A

Элемент области определения A	Элемент области значений A(x)
1	4
2	6
3	3
4	4
5	6
6	8

Мы хотим, чтобы в Z-спецификации было сказано, что t' имеет ту же область определения и те же элементы, что и t, что ее элементы упорядочены (и могут быть равны). Легко выразить факт, что элементы A упорядочены, но нам также придется и учесть повторение элементов. Вспомните, что в математических множествах элементы не повторяются. Например, 6 встречается дважды среди элементов A. Мы должны убедиться, что 6 также встречается дважды среди элементов A'. Для этого мы можем использовать нотацию обратной функции f^{-1}. Значение $f^{-1}(x)$ — это множество элементов области определения, отображающихся на x. Множество элементов из A, отображающихся на 6, — это 1 и 5, так что $A^{-1}(6) = \{1, 5\}$. Нам только осталось убедиться, что $t'^{-1}(6)$ также содержит ровно два элемента, чтобы убедиться в корректности повторений элементов в A и A'. Таким образом, нам нужно проверить, одинаковы ли размеры множеств $A^{-1}(x)$ и $A'^{-1}(x)$ для соответствующих значений x.

Используя Z-нотацию, мы получим спецификацию, показанную на рис. 4.20.

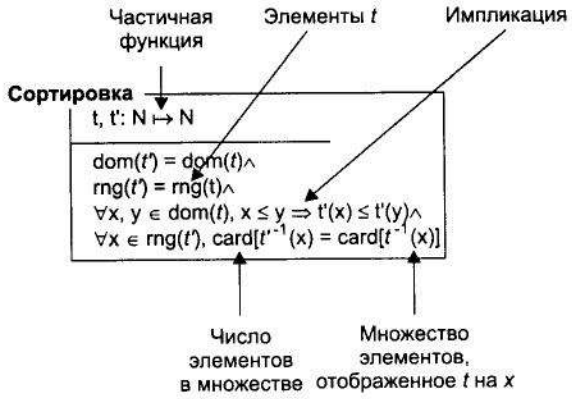


Рис. 4.20. Z-спецификация для примера Сортировка

Пример 4: Максимум

В качестве последнего примера возьмем распространенное простое требование, которое можно сформулировать формально и неформально. На этот раз мы сформулируем его сначала формально (рис. 4.21). Это спецификация максимума массива целых чисел *t* и индекса максимального элемента. Если существует несколько максимальных элементов массива, следует выбрать элемент с наименьшим индексом.

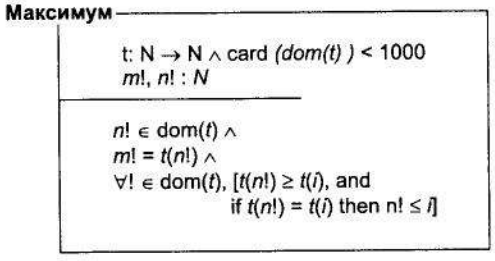


Рис. 4.21. Z-спецификация для максимума

4.8.3. Когда следует использовать формальную спецификацию

Мнения относительно использования формальной спецификации разделились. Несколько коммерческих систем, в том числе широко распространенная система CICS (Customer Information Control System — система управления информацией заказчика) от IBM, использовали формальные методы, в частности Z-спецификации.

Эффективность во взаимодействии требований является показательным тестом, позволяющим решить, формальные или неформальные требования следует

использовать. Выражение требований иногда лучше всего осуществить с помощью естественного языка, а иногда — в виде формальной спецификации. Для того чтобы последнее было эффективно, разработчики и пользователи спецификации требований к программному обеспечению должны быть обучены соответствующим образом. Z-спецификации показали себя наиболее применимыми в процессах, которые можно описать в терминах выходных данных (например, сам алгоритм не нужно описывать). Исследования в области формальных спецификаций продолжаются. К ним также относится изучение использования формальных спецификаций для определения графических пользовательских интерфейсов. Достижения математики трудно не оценить. Вполне возможно, что новые исследования расширят практическую применимость формальных методов.

Короче говоря, если требование можно объяснить в терминах конкретных выходных данных и разработчики обучены формальным методам, формальные методы могут стать существенным средством выражения детальных требований. В любом случае мы можем позаимствовать конкретные части формальных спецификаций, такие как использование обозначений ! и ?.

Формальные спецификации необходимы при реализации *исполняемых спецификаций* (например, [97]). Это спецификации, которые можно автоматически перевести в объектный код. Следовательно, они должны быть четкими и точными. Активные исследования в этой области успешно проводились на протяжении многих лет (например, [87]). Опубликованным примером формальных требований для большого программного приложения реального времени является программа управления полетами для американского флота [A11]. Также следует отметить, что эти детальные требования были написаны в системе изготовителя для демонстрации.

Мы вернемся к применению математики в главе 7, когда будем обсуждать реализацию функций. Сайты [113] также предоставляют современную информацию о Z-спецификациях.

4.8.4. Предусловия и постусловия

Широко используемый тип спецификаций, менее формальный, чем Z-спецификации, состоит из *предусловий* и *постусловий*. Это описание требуемого состояния программы до и после вычислений. Предусловия и постусловия обычно используют псевдокод, однако некоторые их части пишутся на языке реализации (Java, C++ и т. д.). По этой причине они часто используются для определения проектных решений, точнее, они обычно используются для определения требований к функции для программистов. (Вспомните, что, хотя мы называем «требованиями» только одну фазу водопадного процесса, результаты на любой фазе, например на фазе проектирования, становятся требованиями для следующей фазы. В этом случае «реализации требований» — это на самом деле термин, применимый ко многим фазам.)

В качестве примера приведенную ранее задачу нахождения максимума можно сформулировать, как показано на рис. 4.22.

Это требование можно сформулировать более кратко (упражнение О4.4).

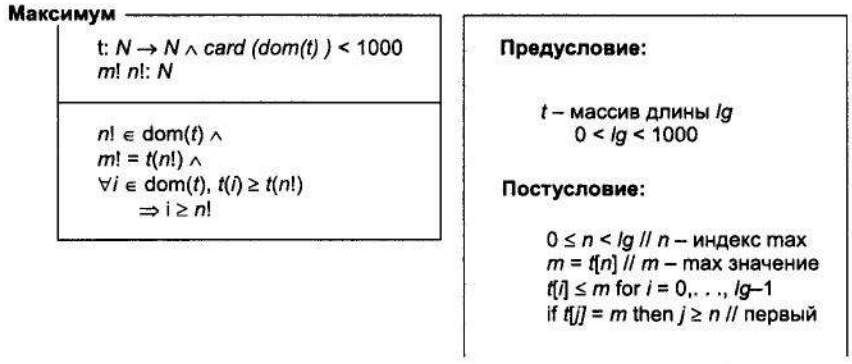


Рис. 4.22. Z-спецификации и предусловия с постусловиями для задачи поиска максимума

4.9. Влияние процесса составления D-требований на проект

После того как D-требования собраны, должны обновляться документы проекта. Как пример рассмотрим необходимые обновления для SPMP.

4.9.1. Влияние на SPMP

Как только все D-требования собраны, SPMP можно обновлять (табл. 4.6). D-требования передаются под управление конфигурациями. Один из вопросов, которые нужно обсудить, — какой уровень детализации будет считаться элементом конфигурации программы. Конечно, таким элементом может быть раздел 3 («Детальные требования») из SRS (в стандарте IEEE). Каждый класс может быть элементом конфигурации. Отдельные требования обычно слишком малы, чтобы выносить их в отдельный элемент конфигурации.

Таблица 4.6. Обновление проекта после завершения D-требований

	Статус после предварительного варианта	Результат обновления C-требований	Результат обновления D-требований
Вехи	Исходный	Более подробны	Более подробны
Риски	Определить	Исключить риск, определенный раньше. Поиск новых факторов риска	Исключить риск, определенный раньше. Идентифицировать новые факторы риска
План-график	Очень высокий уровень	Предварительный план проекта	Более подробный: показывает классы и методы задач разработки

продолжение

Таблица 4.6 (*продолжение*)

	Статус после предварительного варианта	Результат обновления C-требований	Результат обновления D-требований
Персонал	Назначить разработчиков C-требований	Разработчики, назначенные для анализа D-требований	Назначить ответственного за архитектуру программы
Оценка стоимости	Первичные оценки	Первые оценки на основе содержимого работы	Улучшенная оценка на базе более конкретной оценки функционального размера или предыдущего опыта с подобными отдельными требованиями

4.9.2. Влияние размера на D-требования

Когда список требований разрастается до сотен элементов, легко может возникнуть несогласованность, упоминавшаяся в разделе 4.3.6. Классификация требований по классам, классов по пакетам и т. д. становится необходимостью. Группы обычно соответствуют подсистемам в общей организации программного приложения.

Хотя *полнота* — это цель, к которой мы стремимся при сборе требований, она может стать иллюзорной целью. Для существенных программных приложений редко бывает «последнее» требование — оно последнее лишь до момента фиксации требований.

Как было отмечено в соответствующем разделе предыдущей главы относительно масштабов проекта, крупномасштабные проекты требуют более формальной организации (не путать с формальными методами). SRS приходится разбивать на несколько томов. Отдельный раздел в нашем (крошечном!) примере мог бы разрастись до 700 страниц. Для планирования разработки и проверки D-требований необходимо было бы выполнить огромный объем работы. Проекты с сотнями детальных требований нуждаются в инструментах управления требованиями. Опыт успешного широко распространенного применения пакетов Java показал, что большими наборами требований можно управлять, когда функциональность организована четко определенными пакетами и классами.

Вознаграждение за хороший анализ требований существенно. И наоборот, расплата за скудные требования также значительна. Например, в [107] приводится проведенное Правительственным экономическим отделом исследование одного проекта, в котором «проблемы, связанные с требованиями» привели к превышению бюджета на 600 миллионов долларов, задержке в восемь лет и урезанным возможностям. До сих пор ведутся яростные дебаты о проценте крупных неудачных проектов по отношению к проценту успешно завершившихся. Достаточно сказать, что многие крупные проекты уделяют значительное внимание анализу требований. Автор может подтвердить это благодаря личному опыту.

4.10. Подведение итогов процесса определения D-требований

D-требования (требования разработчиков или детальные требования) пишутся преимущественно для проектировщиков и разработчиков. Они создаются из C-требований, а также в результате длительных переговоров с заказчиком. D-требования должны быть тестируемыми, прослеживаемыми и согласованными. Поскольку они разрастаются до больших объемов, их следует систематически классифицировать. Удобным способом организации D-требований является классификация по основным классам предметной области и функциям. Существует несколько метрик, по которым можно проверить D-требования. Хороший анализ требований приносит значительную пользу. Подведем итоги этой главы:

- D-требования предназначены для разработчиков.
- Цели составления D-требований: ясность, прослеживаемость требований.
- Хорошая организация приносит большую пользу.
  Пример — объектно-ориентрированный стиль.
- Следует по возможности использовать формальные методы.
- Результатом является обновление SPMP.

Руководство по учебному проекту. D-требования для примера игры Встреча

В этом разделе объясняется, как принципы получения и представления D-требований, описанные в данной главе, можно применить на практике. Студенту также предлагается обратить внимание на врезки «Один из способов...» в этой главе, поскольку они служат руководством к процессу.

Этап 1. Подготовка

Халл и Карен закончили написание C-требований на основе обсуждений и интервью с Бетти Симз и Арланом Ховардом. Они использовали заголовки стандарта IEEE (раздел 3.1.4) в качестве подсказок для нефункциональных требований, таких как требования, относящиеся к графическому пользовательскому интерфейсу, производительности и аппаратным платформам. Теперь им нужно определить способ организации функциональных D-требований. Они предвидели необходимость многочисленных повторных обращений и исправлений в SRS и хотели упростить этот процесс насколько это возможно. В результате их основным критерием стала возможность легко поддерживать согласованность между SRS, проектированием и кодом.

Сначала они обсудили организацию детальных требований по состояниям и действиям, базируясь на диаграмме переходов состояний, которая была описана

в С-требованиях. Этот способ организации состоял бы из списка действий, предпринимаемых игроком, таких как щелчок на гиперссылке в зоне, после которого должны следовать результаты действия. Они оба согласились, что эта организация была бы понятна, но решили, что она не будет прослеживаться в реализации настолько хорошо, как им бы этого хотелось. Они начали поиск других способов организации D-требований.

Халл выступил за организацию функциональных D-требований по вариантам использования, поскольку он хотел следовать USDP. Он указал, что на этом этапе видеоигру легче всего представлять себе как варианты использования «Инициализировать», «Перейти в соседнюю зону» и «Вступить в контакт с внешним персонажем». Он также отметил, насколько удобно было бы обойтись этими тремя вариантами использования для описания всех функциональных требований. Ему также нравилась мысль о возможности повторного использования этих вариантов для определения будущих игр.

Карен согласилась, что требования будет легче понять, если организовать их по вариантам использования, однако у нее было несколько возражений. Во-первых, некоторые требования будут участвовать более чем в одном варианте использования. В качестве примера она привела ситуацию, когда пользователь щелкает на гиперссылке выхода из комнаты. Это может быть частью всех трех вариантов использования, определенных ими, и будет неясно, где искать конкретное требование. Во-вторых, отображение вариантов использования на код может быть сложнее, чем при организации, задуманной ею. Наконец, Карен отметила, что компания еще не имеет соответствующего архива вариантов использования для будущих работ.

Карен хотела организовать функциональные требования по классам, что, по ее мнению, облегчало прослеживаемость между требованиями и кодом. Она хотела достаточно аккуратно выбрать классы, чтобы убедиться, что они будут использованы при проектировании (и реализации). Халл отметил недостаток такого подхода: тот факт, что им придется довольно рано утвердить несколько классов, которые будут использованы при реализации приложения. Он беспокоился о том, что, возможно, в будущем у них с Карен может измениться мнение относительно набора выбранных классов. После дальнейшего обсуждения они решили, что организация требований по классам выгоднее, и остановились на этом способе. Однако они решили быть очень осторожными в выборе классов.

Этап 2. Классификация D-требований

Сначала Халл и Карен рассмотрели каждый вариант использования и изобразили его на диаграмме последовательности. Изучая каждый шаг, они определили, какой объект какого класса инициирует действие и какой объект отвечает за выполнение действия. Этот процесс помог им создавать и (или) определять классы. Им пришлось несколько раз звонить Бетти и Арлану для уточнения шагов вариантов использования, которые они не совсем четко понимали.

Халл перечислил классы и объекты, упоминавшиеся в вариантах использования. Затем Халл и Карен постарались обсудить каждый аспект игры, какой только могли придумать, на предмет дополнительных возможных классов. На послед-

нем шаге в процессе выбора классов они сильно сократили список, оставив лишь несколько самых важных классов и в то же время стараясь сохранить соответствие всех классов вариантам использования. Получившийся список содержал классы: *Зона*, *ПерсонажВстречи*, *ИграВстреча*, *Контакт*, *ОкноКонтакта*, *ГиперссылкаСоединения*, *ВнешнийПерсонаж*, *ПерсонажИгрока* и *ОкноХарактеристикИгрока*.

Далее они окончательно сформулировали заголовки SRS в разделе 3.2 («Детальные требования»). Детальные требования, соответствующие классу *Зона*, они сгруппировали по зонам игры в подразделе 3.2.A. Эти подразделы они упорядочили по алфавиту, поскольку предвидели добавление классов в будущем. Они предположили, что если упорядочить разделы требований по номерам (например, классу *ПерсонажИгрока* будет соответствовать подраздел номер 3.2.14), то найти конкретное требование будет сложнее, поскольку пользователю SRS придется пролистать многочисленные подразделы 3.2.N, прежде чем он найдет необходимое ему требование. Следующему классу, *СоединениеЗоныВстречи*, они отвели подраздел 3.2.C3B, и т. д. Внутри каждого такого подраздела они создали подразделы для атрибутов, объектов, функциональности и событий.

Этап 3. Написание D-требований

Карен и Халл написали раздел 3.1 о пользовательских интерфейсах, предоставив детали на рисунках, которые они сделали еще для C-требований, и затем попросили Бетти и отдел инженерной психологии просмотреть его. Зная, что это будет финальный документ, по которому будет создаваться программное приложение, они утвердили с заказчиком каждую деталь.

Карен и Халл привели диаграммы последовательности (см. рис. 4.37, рис. 4.35 и рис. 4.28).

Они проверили свои заметки из интервью с Бетти и Арланом на предмет свойств (атрибутов) каждого раздела классификации (класса). Например, они спросили, какие свойства должны быть у соединений между двумя зонами. (Одним свойством такого соединения должна быть *первая зона*, другим *вторая зона*.) Для каждого класса они спросили самих себя, какие объекты этого класса будут необходимы в игре. Например, должны будут существовать объекты *гардероб* и *двор* класса *Зона*. Затем они поинтересовались, какой функциональностью должен обладать класс. Например, функциональность каждого персонажа игры *Встреча* — это возможность изменять значения характеристик (требование 3.2.ПВ.3.2). Наконец, они перечислили все события, на которые должны будут реагировать объекты класса. (Например, щелчок на гиперссылке выхода из зоны.)

Один беспокоивший их аспект заключался во времени, необходимом для того, чтобы новые значения характеристик начинали действовать. Они поняли, что это ключевой момент в игре: если это будет происходить мгновенно, игрок просто будет устанавливать характеристики, соответствующие данной зоне, на максимум, и игра потеряет интерес. Задержка придаст интерес игре, но проблема в том, какова должна быть эта задержка? Халл и Карен сначала хотели отложить решение этой проблемы, но в конце концов решили сделать задержку продолжительностью в четыре секунды, считая, что изменить это число будет достаточно просто.

Карен беспокоилась о влиянии некоторых требований, особенно тех, от которых зависит обмен характеристиками при контакте. Она боялась, что программисты могут не понять эти требования. Это привело бы к потере времени на исправление ошибок либо к некорректно работающей игре. Она предложила использовать Z-спецификацию. Халл заметил, что никто, кроме Карен, не сможет понять эту спецификацию, поскольку остальная часть команды не была этому обучена. В результате они договорились использовать в определении требования подходящий математический аппарат, но не формат Z-спецификации. Карен отметила пор себя, что если бы она преподавала разработку программного обеспечения, она бы настаивала, чтобы студенты свободно пользовались Z-спецификацией.

Используя заголовки разделов из IEEE SRS, Карен и Халл проверили, все ли исполнительные требования они перечислили (в основном требования, относящиеся к скорости игры, что может повлиять на интерес), и предоставили их на рассмотрение Бетти и Арлану. Они также прошлись по ограничениям памяти (оперативная память и диск), после чего завершили работу над документом.

Этап 4. Завершение: метрики и итоги

Группа, анализировавшая требования, попросила Бетти, Арлана и остальных членов команды проверить D-требования. Проверка в основном свелась к сравнению с C-требованиями с целью убедиться, что все моменты C-требований включены в D-требования. Также был составлен список, использованный в табл. 4.3. Было обнаружено несколько недостатков, которые Халл с Карен записали и исправили. Результаты этого процесса оказались похожи на результаты, описанные в руководстве по учебному проекту для C-требований (глава 3).

Упражнения

Ответы и подсказки для упражнений, помеченных символами «о» или «п», приводятся в конце этой главы.

Вопросы для проверки

П4.1°. Для кого преимущественно создаются D-требования?

П4.2°. У типичной программы есть много требований. Назовите важную проблему, сопутствующую созданию требований и работе с ними.

П4.3°. Назовите 3–5 категорий детальных требований.

П4.4°. Назовите 5–7 желаемых свойств детальных требований.

П4.5°. Назовите 4–6 способов организации детальных требований.

П4.6°.

1. Существует ли диаграмма последовательности, соответствующая каждому варианту использования?
2. Существует ли вариант использования, соответствующий каждой диаграмме последовательности?

Общие упражнения

О4.1[11]. Напишите десять D-требований для программы, эмулирующей обслуживание клиентов в банке.

О4.2. Что неправильно в приведенных ниже D-требованиях?

1. Приложение *ДомашнийБюджет* должно предоставлять удобный интерфейс для ввода личных данных.
2. Приложение *УправлениеСпутниками* должно вычислять предполагаемое время, которое займет оборот вокруг Земли по текущей орбите, и время, фактически затраченное для облета Земли по предыдущей орбите.
3. Прогорамма *КорольИнвестиций* должна определять наилучшую стратегию инвестиций.

О4.3. Укажите точные метрики качества D-требований в упражнении О4.2.

О4.4. Сформулируйте требования к вычислению максимума в примере 4 из раздела 4.8.2 в более сжатой форме. Используйте форму предусловий и постусловий.

О4.5. Приведите вариант использования и соответствующую диаграмму переходов состояний для системы со следующим С-требованием:

Система должна давать советы для новичка — пользователя Windows — относительно того, как выполнять операции Windows.

Упражнения в команде

К4.1. (SRS.)

Напишите SRS для вашей программы. Используйте или модифицируйте стандарт IEEE. Если вы используете итеративный процесс разработки, постарайтесь указать, какие требования должны быть реализованы на какой итерации.

Ведите учет времени, затраченного на это, у каждого отдельного человека и у группы. Разбейте все время на соответствующие действия. Измерьте эффективность вашей работы. (Можете разработать свои собственные метрики; обратите также внимание на командные упражнения в предыдущих главах.) Укажите, как ваш процесс разработки спецификации можно было бы улучшить.

Критерии оценки.

1. Уровень четкости («Отлично» — очень четко написано).
2. Степень, в которой план включает все важные детали и не включает неважный материал («Отлично» — содержит более 95 % важных деталей и менее 5 % несущественных деталей).
3. Эффективность вашей самооценки и описание улучшения процесса («Отлично» — очень эффективные метрики; очень конкретные предложения относительно того, как процесс можно было бы улучшить с небольшими временными издержками).

Подсказки

О4.1. Упорядочить по классам. Должны быть добавлены следующие классы: *КлиентБанка*, *Банк*, *Кассир*.

Ответы

П4.1. Преимущественно разработчики; клиенты — вторично.

П4.2. Классифицировать их так, чтобы их можно было легко находить и поддерживать.

П4.3. Функциональные, нефункциональные, обратные, интерфейсные, проектные и ограничения реализации.

П4.4. Прослеживаемость, тестируемость, однозначность, приоритет, полнота, условия ошибок, согласованность.

П4.5. По режиму, действующему лицу варианта использования, классу (объектно-ориентированный стиль), характеристикам, иерархии функций или состояниям.

П4.6.

1. Да. Всегда можно определить затронутые в варианте использования объекты, а затем преобразовать последовательность действий пользователя (системы) в последовательность вызовов функций этих объектов.
2. Нет. Диаграмма последовательности выражает последовательность вызовов функций. Не каждая такая последовательность представляет типичное взаимодействие между пользователем и программой.

Пример. Спецификация требований к программному обеспечению (SRS) для видеоигры Встреча, часть 2

История версий этого документа.

- x/yy/zzz Исходный черновик, написан Карен Петерс.
- x/yy/zzz Техническая точность, проверена Халом Фурнессом; требования классов *Зона* и *ПерсонажИгрока* изменены.
- x/yy/zzz Версия 0.1 принята Карен Петерс.

3. Детальные требования

3.1. Требования к внешнему интерфейсу

3.1.1. Пользовательские интерфейсы

[Примечание для студентов. Раздел 2.1.2 в SRS видеоигры Встреча содержит лишь наброски пользовательских интерфейсов для общего представления продукта. В них не хватает деталей, и они не должны рассматриваться как окончательные.

Если пользовательские интерфейсы не будут полностью определены далее в этом документе, все детали для этого должны быть даны в этом разделе. Поскольку в этом примере мы используем объектный стиль спецификации, подробности каждого окна находятся в соответствующих классах в разделе 2.2 SRS.

В любом случае этот раздел должен объяснить физическую связь между графическими элементами (например, расположение каскадом, и т. д.).]

Действие игры *Встреча* происходит в зонах. Типичный экранный снимок зоны *двор* с персонажем, контролируемым игроком, внешним персонажем и результатами контакта показан на рис. 4.23. Этот интерфейс занимает весь экран монитора. Зоны соединены с соседними зонами. Эти соединения обозначены гиперссылками. Щелчок на одной из гиперссылок переносит персонаж игрока в соответствующую зону.

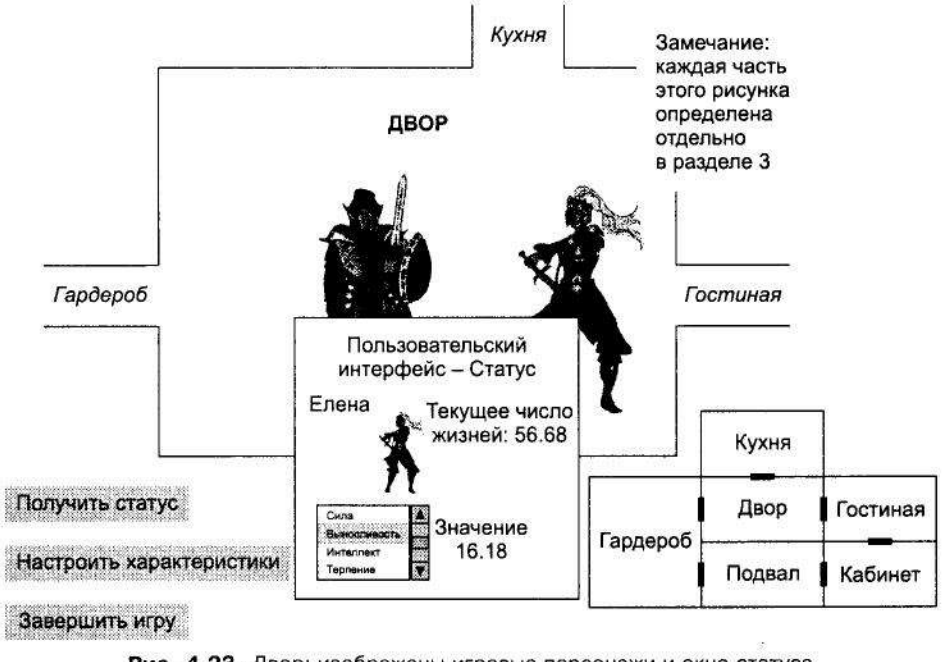


Рис. 4.23. Двор: изображены игровые персонажи и окно статуса

Вот полный набор интерфейсов.

1. Один пользовательский интерфейс для каждой зоны, определенной в разделе 3.2.З ниже.
2. Пользовательский интерфейс, позволяющий настроить значения характеристик персонажа игрока, определен в разделе 3.2.ХИ.
3. Пользовательский интерфейс для показа результатов контакта, определенный в разделе 3.2.КД. Тот же пользовательский интерфейс используется для показа статуса персонажа игрока.

Интерфейс типа 1 всегда будет присутствовать на мониторе. В случаях, определенных этими требованиями, интерфейсы типов 2 и 3 будут показаны поверх первого. Это требование зафиксировано в STD (здесь должна быть ссылка на тест).

3.1.2. Аппаратные интерфейсы

[Примечание для студентов. Оборудование, на котором будет работать игра Встреча.]

Нет.

[Будущие версии: для игры можно будет использовать джойстик.]

3.1.3. Программные интерфейсы

[Примечание для студентов. Другие программы, с которыми игра должна взаимодействовать, например драйвер принтера.]

Нет.

[Будущие версии: в игру можно будет играть с сайта Intergalactic Internet Gaming Site.]

3.1.4. Коммуникационные интерфейсы

Нет.

[Будущие версии: игра *Встреча* будет иметь интерфейс для выхода в Интернет посредством модема с минимальной скоростью 56 Кбайт/с.]

3.2. Детальные требования

[Примечание для студентов. В этом разделе вы располагаете некоторой свободой относительно стандартов IEEE, чтобы учесть варианты использования. Сначала здесь представлены диаграммы последовательности, необходимые для описания вариантов использования из раздела 2.2 данной SRS (раздел 3.2.1). Классы, необходимые для описания этих вариантов использования, затем применяются для классификации детальных требований (раздел 3.2.2 SRS). Диаграммы последовательности объяснены в разделе 4.4 этой главы.]

3.2.1. Диаграммы последовательности

[Примечание для студентов. Здесь мы покажем диаграммы последовательности для каждого варианта использования, определенного в разделе 2.1 данной SRS. В требованиях стандарта IEEE нет раздела «Диаграммы последовательности»; он добавлен для удобства.]

3.2.1.1. Вариант использования «Инициализировать»

Диаграмма последовательности для варианта использования «Инициализировать» показана на рис. 4.24. Для этого варианта использования необходимы такие классы, как *ИграВстреча* (у которого будет только один объект), *ПерсонажИгрока* (объектом которого будет главный персонаж игрока), *ОкноХарактеристикИгрока* (у которого будет только один объект) и *Зона* (объектом которого будет *гардероб*).

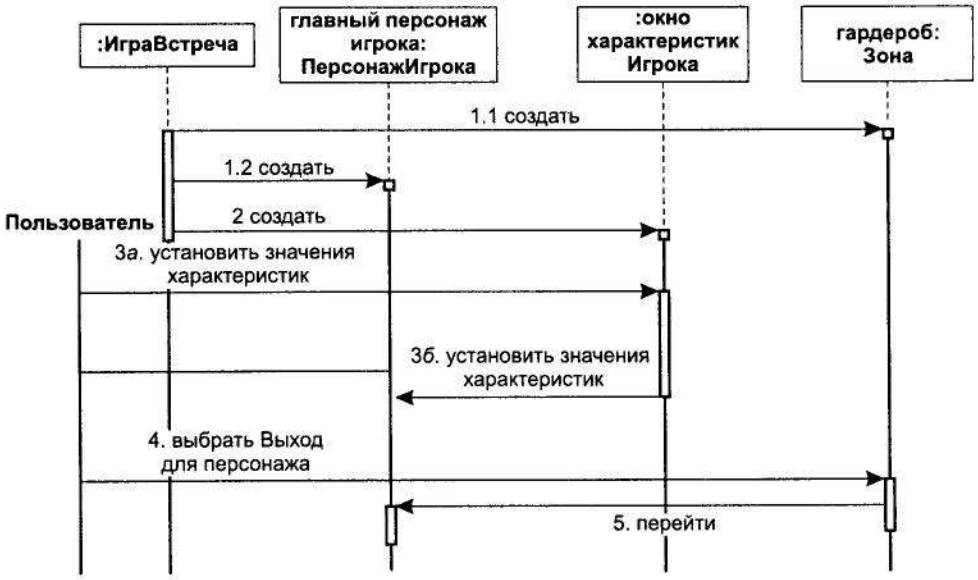


Рис. 4.24. Диаграмма последовательности для варианта использования «Инициализировать»

3.2.1.2. Вариант использования «Перейти в соседнюю зону»

Диаграмма последовательности для варианта использования «Перейти в соседнюю зону» изображена на рис. 4.25. Для этого варианта использования необходимы классы *ГиперссылкаСоединения*, *СоединениеЗоны* и *ПерсонажИгрока*.

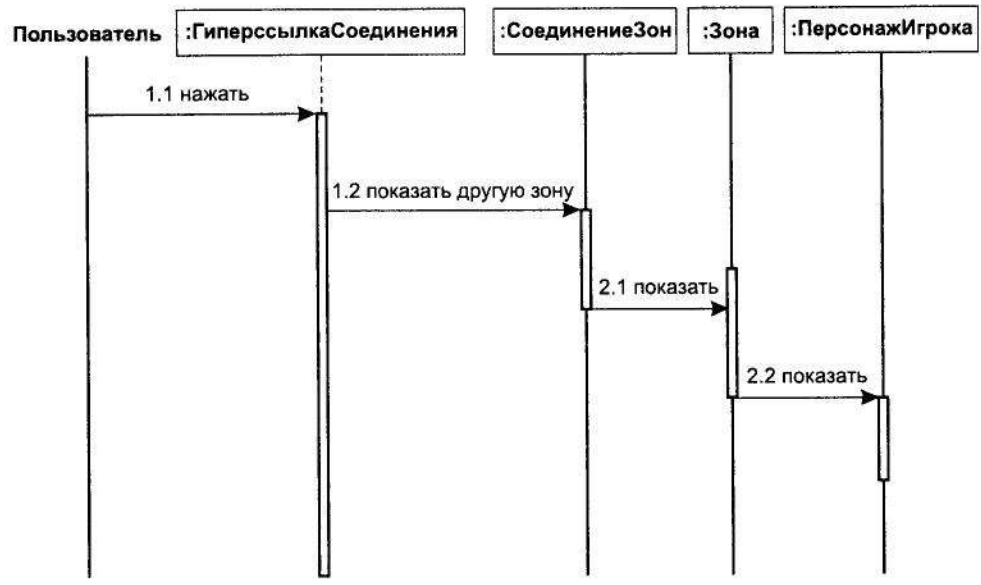


Рис. 4.25. Диаграмма последовательности для варианта использования «Перейти в соседнюю зону»

3.2.1.3. Вариант использования «Вступить в контакт с внешним персонажем»

Диаграмма последовательности для варианта использования «Вступить в контакт с внешним персонажем» показана на рис. 4.26. Для этого варианта использования необходимы классы *ИграВстреча* (у которого будет только один объект), *ВнешнийПерсонаж* (например, объект *Фредди*), *Контакт*, *ПерсонажИгрока*, *ОкноХарактеристикИгрока* и *ОкноКонтакта*.

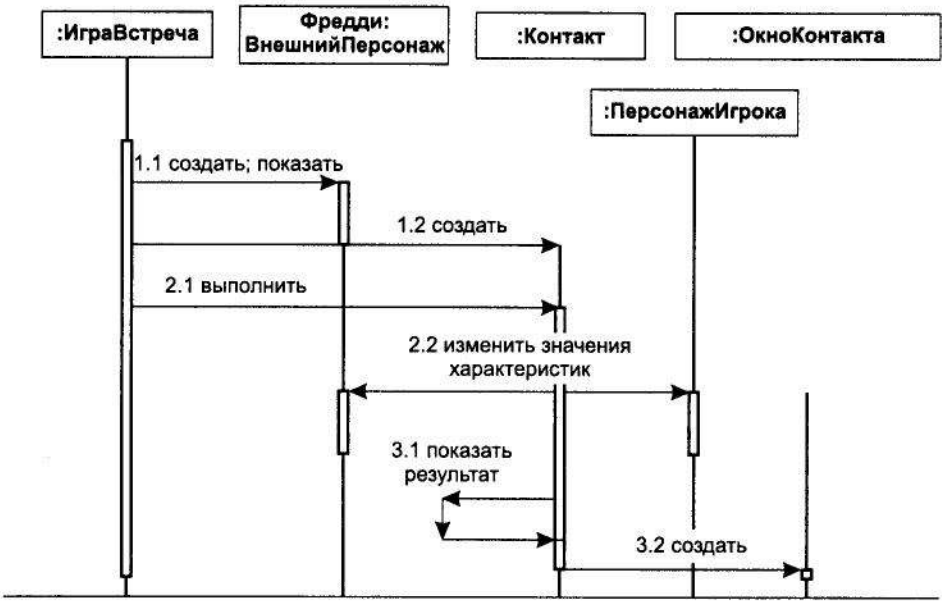


Рис. 4.26. Диаграмма последовательности для варианта использования «Вступить в контакт с внешним персонажем»

3.2.2. Классы для классификации детальных требований

[Примечание для студентов. Поскольку мы упорядочиваем детальные требования по классам, прежде всего мы должны перечислить имеющиеся выбранные классы. Сюда относятся не все классы, которые будут использоваться в программе, а так называемые основные классы, принадлежащие классам предметной области программы. Эти классы удобны для организации всех требований. В данном случае, например, все они являются аспектами видеоигры Встреча.]

Для выражения требований достаточно следующих классов видеоигры *Встреча*: *Зона*, *ПерсонажВстречи*, *ИграВстреча*, *ОкноКонтакта*, *ВнешнийПерсонаж*, *ПерсонажИгрока*, и *ОкноХарактеристикИгрока*. Они показаны на объектной модели (рис. 4.27).

[Примечание для студентов. Нумерация «3.2.Зона.N.N...» и т. д., используемая в разделе 3.2, упрощает операции добавления, удаления и поиска требований путем упорядочения классов, содержащих требования, по алфавиту. Представьте себе, что у вас есть сотни требований. Если мы будем нумеровать классы «3.2.1...»,

«3.2.2...» и т. д., то добавлять новые классы придется в конец списка, поскольку на имеющуюся нумерацию уже будут ссылки в разных местах проекта и ее нельзя будет менять. Требования в этом случае не будут упорядочены по алфавиту. В результате при поиске конкретного требования вам придется читать весь список подряд.]

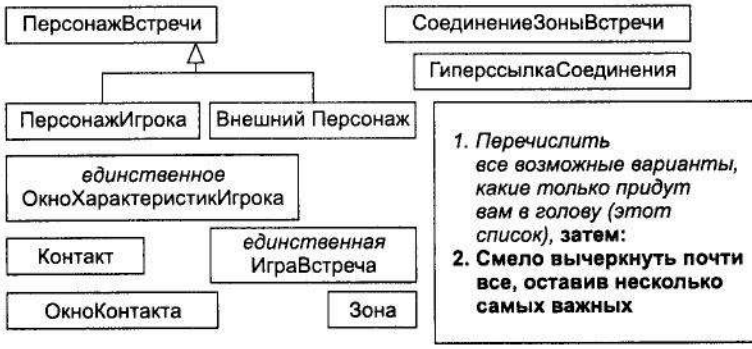


Рис. 4.27. Классы для видеоигры Встреча с отмеченным отношением наследования

3.2.ВП. Внешние персонажи

Внешний персонаж — это персонаж игры *Встреча*, не управляемый игроком.

3.2.ВП.1. Атрибуты внешних персонажей

См. требования к персонажу *Встречи*. Требования для этого класса должны быть теми же.

[В будущих версиях внешние персонажи могут видоизменяться.]

3.2.ВП.2. Объекты — внешние персонажи

[Примечание для студентов. В этом разделе указано, что будет существовать только один внешний персонаж.]

3.2.ВП.2.1. Фредди — внешний персонаж [важно; еще не реализовано]

[Примечание для студентов. Утверждение в скобках выше указывает приоритет и статус требования. Как только требование будет закодировано и протестировано, фразу «еще не реализовано» можно будет убрать или заменить на «реализовано».

Требования, отмеченные словом «важно», должны быть реализованы прежде всех остальных. Как только проект требования для реализации разработан, слово «важно» можно убрать. Это одна из техник прослеживания состояния программы и ее связи с SRS. Другая техника заключается в определении итерации, к которой относится требование.]

Должен существовать внешний персонаж по имени Фредди (рис. 4.28). Этот персонаж изначально будет иметь 100 очков-жизней, распределенных равномерно по всем его характеристикам.

Рис. 4.28. Изображение внешнего персонажа Фредди

3.2.ВП.3. Функциональность внешних персонажей

3.2.ВП.3.1. Движение внешних персонажей [важно; еще не реализовано]

Пока внешний персонаж жив, он должен перемещаться из одной зоны в другую, соседнюю, через случайные промежутки времени со средней длительностью две секунды. После присутствия в зоне в течение случайного промежутка времени со средним значением в одну секунду все очки-жизни персонажа распределяются между характеристиками, специфичными для зоны, так чтобы значения каждой из таких характеристик были по возможности равными.

3.2.ГС. Гиперссылки соединения между зонами

Гиперссылки соединения — это гиперссылки, расположенные на выходе из каждой зоны и показывающие зону, в которую они ведут.

3.2.ГС.1. Атрибуты гиперссылок соединения

3.2.ГС.1.1. Соединение [важно; еще не реализовано]

Каждой гиперссылке соединения соответствует зона, на которую она ссылается.

3.2.ГС.2. Объекты гиперссылки соединения [важно; еще не реализовано]

Существует две гиперссылки соединения, соответствующие соединению между двумя зонами, по одной в каждой соединяемой зоне.

3.2.ГС.3. Функциональность гиперссылок соединения

Нет.

3.2.ГС.4. События, относящиеся к гиперссылкам соединения

3.2.ГС.4.1. Пользователь щелкает на гиперссылке соединения

В результате щелчка на гиперссылке соединения персонаж игрока должен быть показан в зоне, на которую указывает гиперссылка.

3.2.ЗО. Зоны

[Примечание для студентов. Сначала мы опишем, на что ссылается класс (например, данная классификация требований).]

Зона — это видимая область, показываемая на мониторе. Все действия игры (в том числе и контакты) происходят в зонах. Примерами зон могут быть комнаты, сады, дворы.

3.2.ЗО.1. Атрибуты зоны

[Примечание для студентов. Здесь мы расскажем, какими свойствами должен обладать каждый объект этого класса.]

3.2.ЗО.1.1. Название зоны [важно; еще не реализовано]

У каждой зоны будет уникальное название, включающее от 1 до 15 символов. Допустимыми символами будут считаться только пробелы, цифры 0–9, буквы от *a* до *z* и от *A* до *Z*.

План тестирования (здесь должна быть ссылка на тест).

[Примечание для студентов. Каждое требование-атрибут отображается на пару функций get- и set-. Этот документ предлагает способ, которым каждое требование может ссылаться на модульный тест в SDD посредством гиперссылки.]

3.2.ЗО.1.2. Рисунок зоны [важно; еще не реализовано]

Для изображения каждой зоны на мониторе будет существовать отдельный рисунок. Этот рисунок должен быть размером во весь экран.

3.2.ЗО.1.3. Зона — особые характеристики [важно; еще не реализовано]

Для особых характеристик зоны должны применяться лишь некоторые характеристики персонажа. Особые характеристики, требуемые для каждой зоны, определены в разделе 3.2.ЗО.2.

3.2.ЗО.1.4. Кнопки [важно; еще не реализовано]

В каждой зоне должны показываться кнопки Показать статус, Настроить характеристики и Закончить игру в левом нижнем углу.

3.2.ЗО.2. Объекты — зоны

[Примечание для студентов. Мы определим конкретные объекты — зоны, которые должны существовать в программе.]

3.2.ЗО.2.1. Зона «двор» [важно; еще не реализовано]

Должна существовать зона «двор», особыми характеристиками которой будут *выносливость* и *сила*. Предварительное изображение двора с прилагающейся картой соседних зон показано на рис. 4.29.

3.2.ЗО.2.2. Зона «гардероб» [важно; еще не реализовано]

Должна существовать зона «гардероб», не имеющая никаких особых характеристик. Ее предварительное изображение (рис. 4.30) содержит карту соседних зон.

3.2.ЗО.2.3. Зона «подвал» [важно; еще не реализовано]

Должна существовать зона «подвал», особыми характеристиками которой будут *выносливость* и *терпение*. Ее предварительное изображение (рис. 4.31) содержит карту соседних зон.

3.2.ЗО.2.4. Зона «кухня» [важно; еще не реализовано]

Должна существовать зона «кухня», особой характеристикой которой будет *сосредоточенность*. Ее предварительное изображение (рис. 4.32) содержит карту соседних зон.

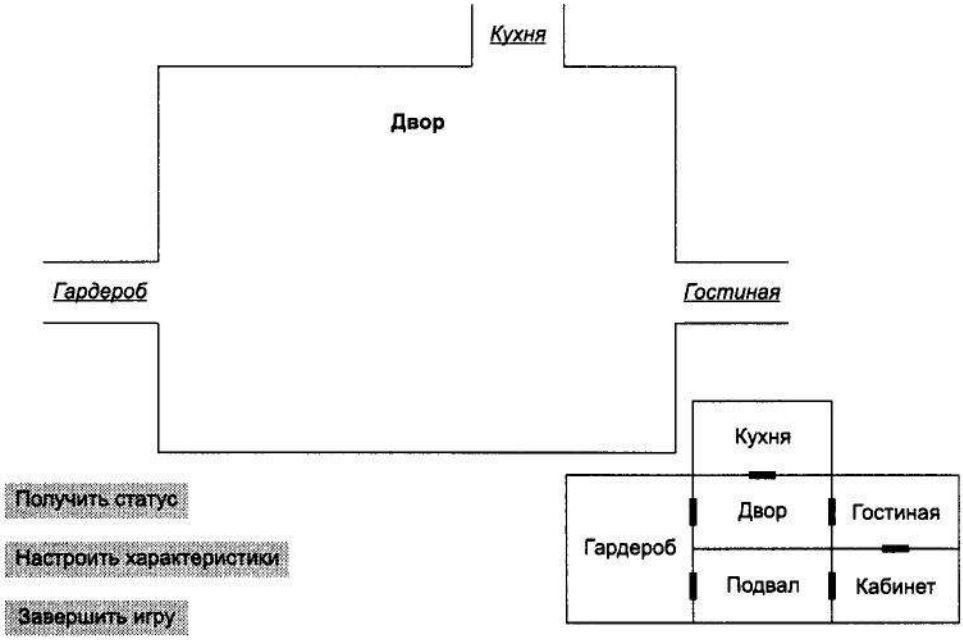


Рис. 4.29. Изображение двора в игре Встреча

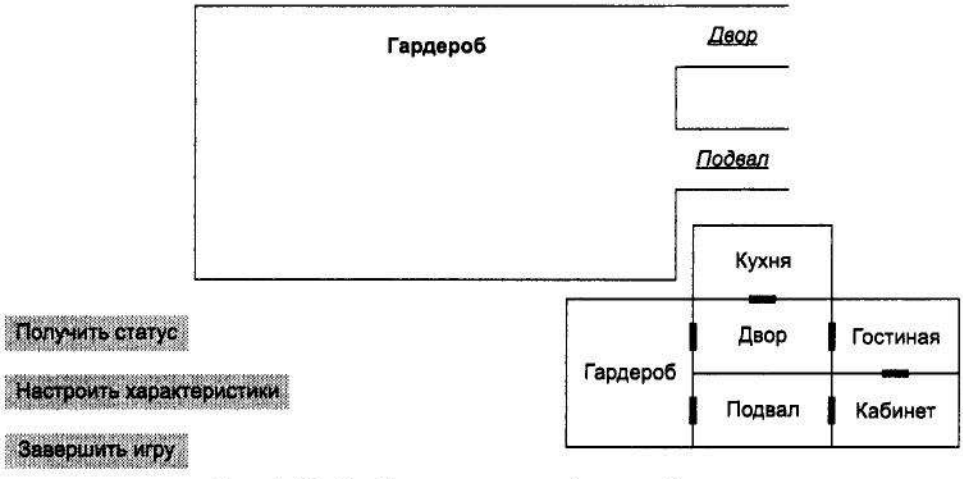


Рис. 4.30. Изображение гардероба в игре Встреча

3.2.ЗО.2.5. Зона «гостиная» [важно; еще не реализовано]

Должна существовать зона «гостиная», особыми характеристиками которой будут *выносливость* и *сосредоточенность*. Ее предварительное изображение (рис. 4.33) содержит карту соседних зон.

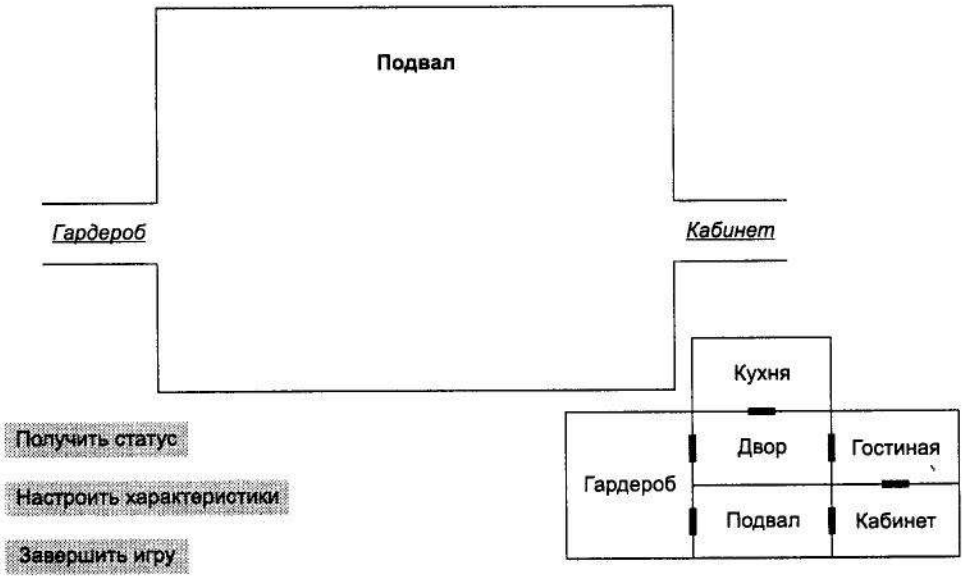


Рис. 4.31. Изображение подвала в игре Встреча

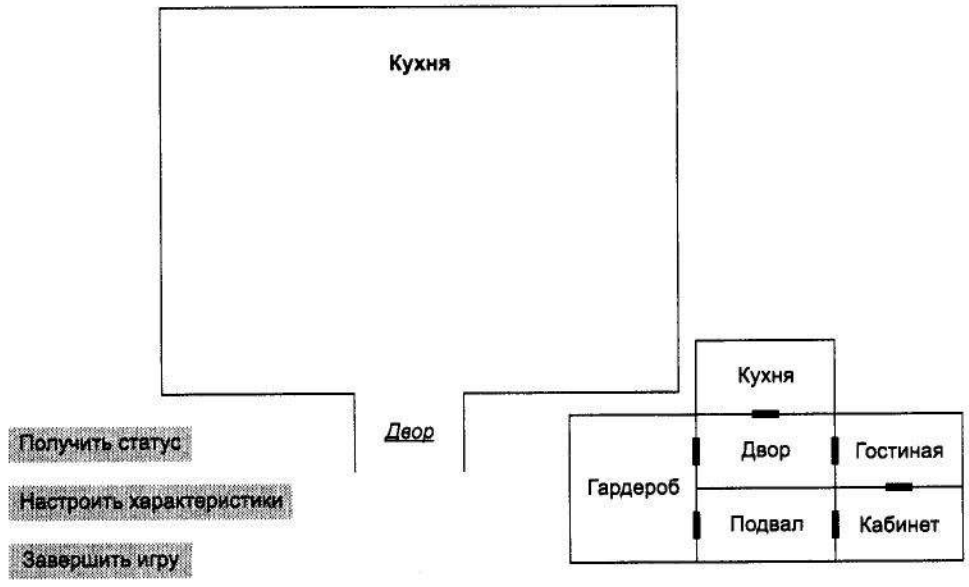


Рис. 4.32. Изображение кухни в игре Встреча

3.2.ЗО.2.3. Зона «кабинет» [важно; еще не реализовано]

Должна существовать зона «кабинет», особой характеристикой которой будет *сосредоточенность*. Ее предварительное изображение (рис. 4.34), содержит карту соседних зон.

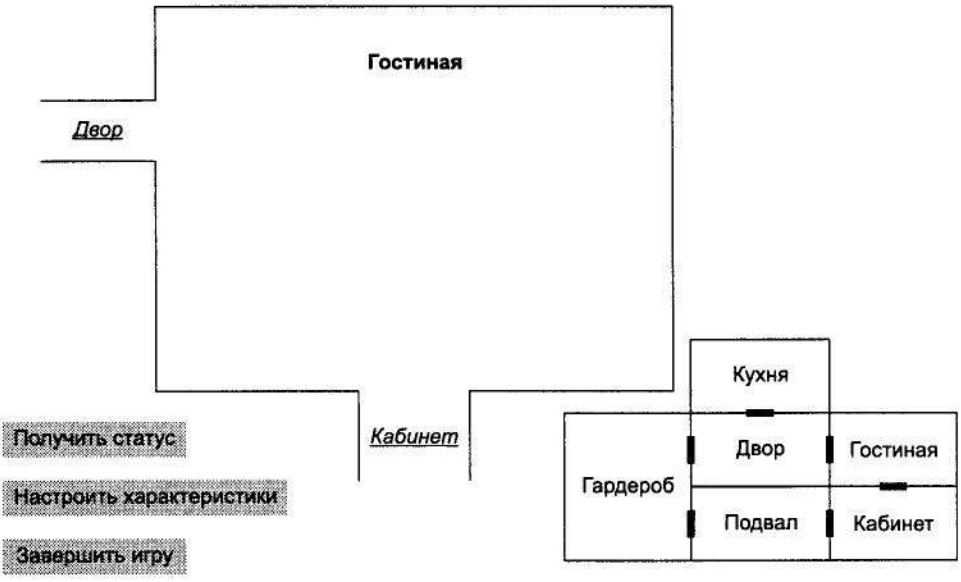


Рис. 4.33. Изображение гостиной в игре Встреча

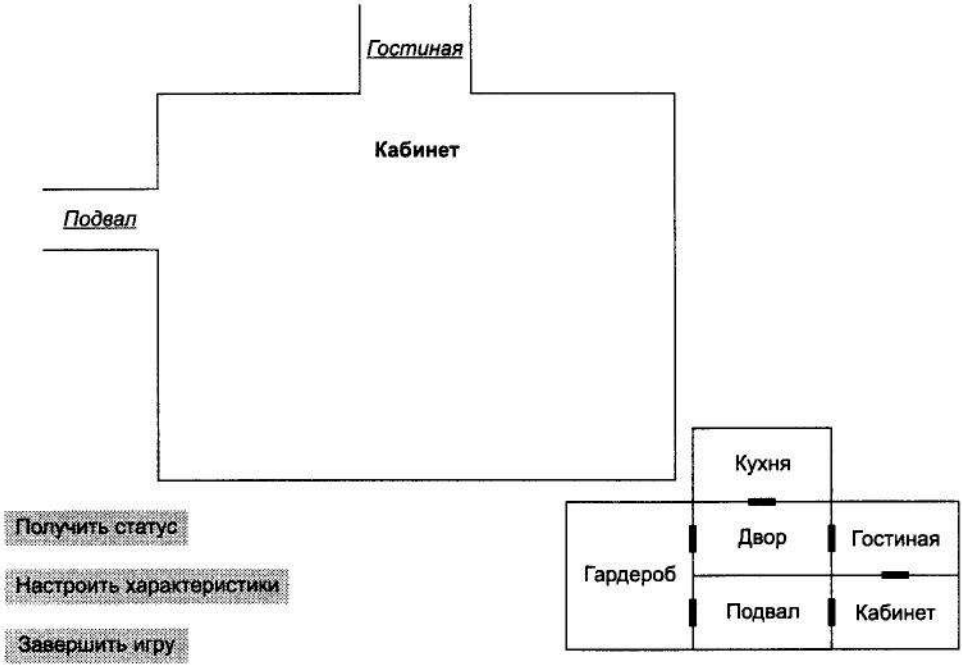


Рис. 4.34. Изображение кабинета в игре Встреча

3.2.ЗО.3. Функциональность зон

[Примечание для студентов. Это обязательная функциональность, свойственная только зонам. Каждая функциональная возможность программы должна относиться к одному из этих разделов.]

Нет.

3.2.ЗО.4. События, относящиеся к зонам

[Примечание для студентов. Мы отделяем события, относящиеся к зонам, от атрибутов, объектов и методов. Событие — это действие извне, на которое программа реагирует.]

3.2.ЗО.4.1. Изображение при входе персонажа [важно; еще не реализовано]

Каждый раз, когда главный персонаж игрока вступает в зону, на экране необходимо показать соответствующую зону и персонажи, находящиеся в ней.

3.2.ЗО.4.2. Обработка контактов [важно; еще не реализовано]

Когда внешний игровой персонаж входит в зону, в которой находится главный персонаж игрока, или наоборот, они вступают в контакт друг с другом.

3.2.ЗО.4.3. Прерывание контактов [не обязательно; еще не реализовано]

Игроки имеют возможность прерывать контакт случайным образом. В среднем игрок может остановить один из десяти контактов путем запуска процедуры настройки свойств. Пользователь пытается прервать контакт, устанавливая характеристики своего персонажа. Если игра не позволяет это сделать, ничего не произойдет: игра продолжается так, как будто попытки изменить характеристики не было вообще.

3.2.ЗО.4.4. Щелчок на кнопке Настроить характеристики [не обязательно; еще не реализовано]

Когда пользователь щелкает на кнопке Настроить характеристики, поверх зоны появляется окно для настройки значений характеристик. Это происходит лишь в том случае, если в зоне нет внешних персонажей. Подробное описание этого окна — в разделе 3.2.ХИ.

3.2.ЗО.4.5. Щелчок на кнопке Закончить игру [не обязательно; еще не реализовано]

Когда пользователь щелкает на кнопке Закончить игру, игра завершает свою работу. Не выводится никаких дополнительных экранов.

[Примечание для студентов. Предыдущее предложение, являющееся обратным требованием, необходимо, поскольку часто игры перед выходом показывают итоги сессии.]

3.2.ЗО.4.6. Щелчок на кнопке Получить статус [не обязательно; еще не реализовано]

Когда пользователь щелкает на кнопке Получить статус, должно появиться окно контакта, в котором должен быть показан статус персонажа игрока до и после последнего контакта.

3.2.ИВ. Игра Встреча

Требования в этом разделе относятся к игре в целом.

3.2.ИВ.1. Атрибуты игры

3.2.ИВ.1.1. Продолжительность [не обязательно; еще не реализовано]

Должна храниться запись с информацией о продолжительности каждой игры, с отсчетом времени с момента, когда игрок запускает игру.

3.2.ИВ.2. Объекты класса Игра Встреча

3.2.ИВ.2.1. Одиночная игра [важно; еще не реализовано]

Игра должна существовать в одном экземпляре.

[Примечание для студентов. В будущих версиях будет возможен одновременный запуск нескольких экземпляров игры.]

3.2.КО. Контакты

Контакт — это взаимодействие между игровым персонажем, управляемым игроком, и внешним персонажем.

3.2.КО.1. Атрибуты Контактов

Нет.

3.2.КО.2. Объекты Контакты

Не будет существовать постоянных объектов этого класса.

3.2.КО.3. Функциональность контактов

3.2.КО.3.1. Вступление в контакт с внешним персонажем [важно; еще не реализовано]

[Примечание для студентов. Это требование по природе своей является математическим, так что здесь не предпринимается попыток заменить математику естественным языком, что могло бы привести к неточности формулировки. Однако использование естественного языка для объяснения математики приветствуется.]

Когда происходит контакт, *сильнейшим* из двух персонажей считается тот, у которого сумма значений характеристик, определяемых зоной контакта, больше. Система передает половину значений каждой характеристики, определенной зоной, от слабейшего сильнейшему. Передачи значений не происходит, если значения персонажей равны по этим характеристикам.

Если у одного из персонажей не остается очков-жизней после перераспределения значений характеристик, игра завершается. Если игра не завершается,

персонаж игрока перемещается в произвольную зону и игроку показываются результаты контакта.

В качестве примера перераспределения значений предположим, что персонаж игрока вступает в контакт с внешним персонажем в зоне, предпочтительными характеристиками которой являются *выносливость* и *сосредоточенность*. Пусть p_s — значение выносливости игрока. Предположив, что $p_s + p_c > f_s + f_c$, получим $p'_s = p_s + f_s / 2, p'_c = p_c + f_c / 2, f'_s = f_s / 2, f'_c = f_a / 2$, где штрихом отмечены значения после контакта. (Внимательный читатель найдет недостаток в предыдущем уравнении, где должно быть $f'_c = f_c / 2$. Оставим этот дефект нетронутым в качестве примера.)

Рассмотрим числовой пример в этой зоне: пусть значение выносливости игрока равно 7, сосредоточенности — 19, а значения выносливости и сосредоточенности Фредди соответственно 11 и 0,6, то есть игрок сильнее. Результаты контакта будут такими:

Игрок: уверенность 7 + 11/2 = 12,5; сосредоточенность 19 + 0,6 / 2 = 19,3

Фредди: уверенность 11/2 = 5,5; сосредоточенность 0, поскольку 0,6 / 2 меньше 0,5.

3.2.ОК. Окно контакта [важно; еще не реализовано]

Должно существовать окно, показывающее результат контакта. Его формат дан на рис. 4.35.

Рис. 4.35. Пользовательский интерфейс для изображения статуса

3.2.ОК.4. События, относящиеся к окну контакта

3.2.ОК.4.1. Закрытие окна контакта [важно; еще не реализовано]

Когда пользователь щелкает на кнопке OK, окно должно исчезать.

3.2.ПВ. Персонаж Встречи

3.2.ПВ.1. Атрибуты Персонажей Встречи

3.2.ПВ.1.1. Имя [важно; еще не реализовано]

У каждого персонажа в видеоигре *Встреча* будет уникальное имя. Ограничения на имена персонажей будут те же, что и для названий зон. Они определены в разделе 3.2.ЗО.1.

3.2.ПВ.1.2. Характеристики [важно; еще не реализовано]

У каждого игрового персонажа имеется одинаковый набор характеристик. Каждая характеристика представлена неотрицательным числом с десятичной запятой и как минимум одной цифрой в дробной части. Все они одинаково инициализируются, так что сумма значений всех характеристик равна 100. Значение характеристики не может быть в промежутке 0–0,5.

В первой версии будут такие характеристики, как *сосредоточенность*, *ум*, *терпение*, *выносливость* и *сила*.

3.2.ПВ.1.3. Изображение персонажа [важно; еще не реализовано]

Каждый персонаж игры будет представлен картинкой.

3.2.ПВ.2. Объекты персонажей

Персонажи игры описаны среди типов персонажей игры *Встреча*.

3.2.ПВ.3. Функциональность персонажей

3.2.ПВ.3.1. Очки-жизни [важно; еще не реализовано]

Игра *Встреча* будет иметь возможность рассчитать сумму значений характеристик любого персонажа. Эта сумма будет называться *числом очков-жизней*.

3.2.ПВ.3.2. Возможность настройки характеристик персонажа [важно; еще не реализовано]

Когда игровой персонаж находится в зоне один, пользователь может изменять значения характеристик. Выбранные значения не должны превышать сумму значений всех характеристик. Значения остальных характеристик автоматически настраиваются так, чтобы сохранить соотношения между ними. Исключением является случай, когда значения остальных характеристик меньше единицы — тогда они заменяются на нули.

3.2.ПИ. Персонажи игрока

Будут существовать персонажи *Встречи*, управляемые игроком.

3.2.ПИ.1. Атрибуты персонажей игрока

См. атрибуты персонажа *Встречи*. Изображение персонажа игрока может быть выбрано из трех (рис. 4.36).

Рис. 4.36. Варианты выбора персонажа игрока

3.2.ПИ.2. Объекты — Персонажи игрока

3.2.ПИ.2.1. Главный персонаж игрока [важно; еще не реализовано]

Игрок будет управлять некоторым конкретным игровым персонажем, называемым *главным* персонажем. Природа этого управления подчинена ограничениям, перечисленным в следующих требованиях. Этот персонаж будет изначально иметь 100 очков-жизней, равномерно распределенных между характеристиками персонажа.

3.2.ПИ.2.2. Дополнительные персонажи, управляемые игроком [не обязательно; еще не реализовано]

У игрока будет возможность заводить другие персонажи помимо главного. Подробности еще не определены.

3.2.ПИ.3. Функциональность персонажей игрока

3.2.ПИ.3.1. Настройка значений характеристик персонажа игрока [важно; еще не реализовано]

Когда в зоне, в которой находится главный персонаж игрока, нет внешних персонажей, игрок может настраивать значения любых характеристик своего главного персонажа, используя *ОкноХарактеристикИгрока* (рис. 4.38). Выбранное значение не должно превышать суммы значений всех характеристик. Значения остальных характеристик автоматически настраиваются так, чтобы сохранять соотношения между ними, за исключением тех характеристик, значения которых получаются меньше 0,5 — в этом случае им будет присвоено нулевое значение.

3.2.ПИ.3.2. Настройка изображения персонажа игрока [желательно; еще не реализовано]

У игрока будет возможность выбрать внешний вид представляющего его персонажа как минимум из двух вариантов (см. рис. 4.36).

3.2.ПИ.3.3. Вычисление возраста персонажа игрока [не обязательно; еще не реализовано]

У главного персонажа игрока значение каждой характеристики будет автоматически увеличиваться на определенный процент во время первой половины его жизни и уменьшаться на тот же процент во второй половине его жизни. Подробности нужно придумать.

3.2.СВ. Соединения между зонами

Персонажи переходят из зоны в соседнюю зону посредством соединений. Каждое из них соединяет две зоны. Необходимые соединения зон показаны на рис. 4.37.

3.2.СВ.1. Атрибуты соединений

3.2.СВ.1.1. Первая и вторая зоны [важно; еще не реализовано]

Каждое соединение будет связывать пару зон, которые мы будем называть *первой* и *второй* зоной.

Рис. 4.37. Конфигурация зон в игре Встреча (желательное требование)

3.2.СВ.2. Объекты — соединения

3.2.СВ.2.1. Гардероб — двор [важно; еще не реализовано]

Будет существовать соединение между гардеробом и двором.

3.2.СВ.2.2. Подвал — кабинет [важно; еще не реализовано]

Будет существовать соединение между подвалом и кабинетом.

3.2.СВ.2.3. Кабинет — гостиная [важно; еще не реализовано]

Будет существовать соединение между кабинетом и гостиной.

3.2.СВ.2.4. Двор — гостиная [важно; еще не реализовано]

Будет существовать соединение между двором и гостиной.

3.2.СВ.2.5. Гостиная — подвал [важно; еще не реализовано]

Будет существовать соединение между гостиной и подвалом.

3.2.СВ.2.6. Двор — кухня [важно; еще не реализовано]

Будет существовать соединение между двором и кухней.

3.2.СВ.3. Функциональность соединений

Нет.

3.2.СВ.4. События, соответствующие соединениям

3.2.СВ.4.1. Перемещение персонажа по соединению [важно; еще не реализовано]

Соединения показываются как гиперссылки на границах зон, когда персонаж находится внутри зоны. Когда пользователь щелкает на такой гиперссылке, показывается соответствующая зона и персонаж в ней.

3.2.ХИ. Окно характеристик игрока

Это окно, в котором пользователь может изменять значения характеристик своих персонажей.

3.2.ХИ.1. Атрибуты окна характеристик игрока

Окно для настройки значений характеристик персонажа игрока в игре *Встреча* показано на рис. 4.38. В центре показывается изображение персонажа игрока,

а его имя — в левом верхнем углу экрана. Очки-жизни персонажа показываются посередине. Слева находится список, показывающий одновременно четыре характеристики. Объяснение математических вычислений показано в бледно-желтом окне в нижней половине экрана. Цвет фона для имени, очков-жизней и окон со значениями должен быть бледно-бирюзовым.

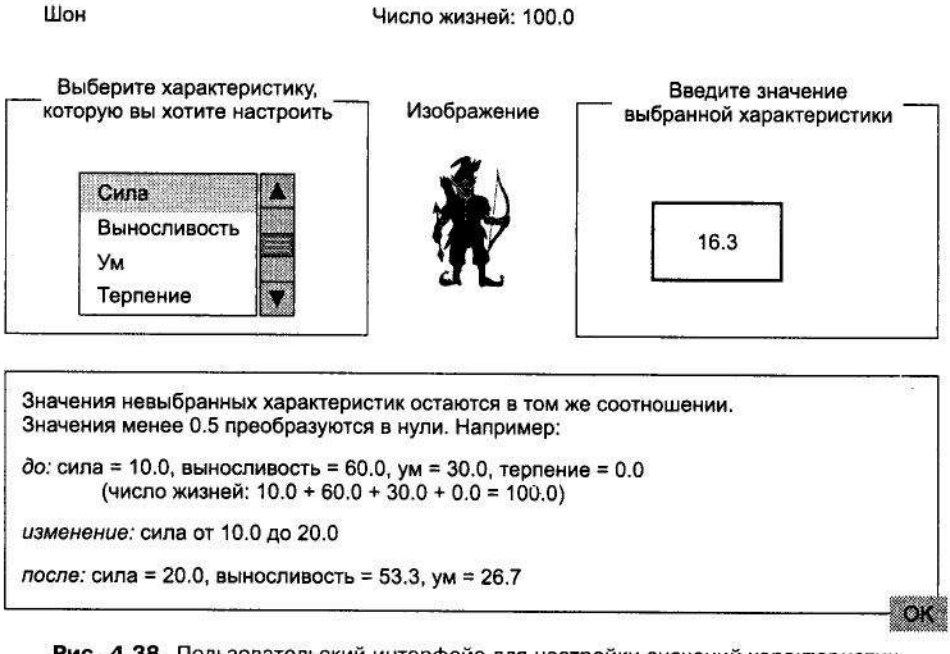


Рис. 4.38. Пользовательский интерфейс для настройки значений характеристик

3.2.ХИ.2. Объекты — окна характеристик игрока

3.2.ХИ.2.1. Окно для назначения характеристик [важно; еще не реализовано]

Окно должно быть доступно при условиях, описанных выше, для настройки значений характеристик персонажа игрока. Окно будет выглядеть согласно описанию, приведенному в разделе 3.1.1.2 этой спецификации.

3.2.ХИ.3. Функциональность характеристик игрока

3.2.ХИ.3.1. Инициализация [важно; еще не реализовано]

Меню характеристик игрока должно иметь возможность показывать себя.

3.2.ХИ.4. События, относящиеся к окну характеристик игрока

3.2.ХИ.4.1. Изображение значения характеристики [важно; еще не реализовано]

Когда игрок выбирает характеристику из списка, расположенного слева, значение этой характеристики должно быть показано в текстовом окне справа.

3.2.ХИ.4.2. Изменение значения характеристики [важно; еще не реализовано]

Когда пользователь вводит допустимое значение характеристики и нажимает клавишу Enter, значение характеристики устанавливается равным введенному числу. Если введено неразрешенное значение, должно появиться окно с сообщением об ошибке «Недопустимое значение: попробуйте еще раз».

3.2.ХИ.4.3. Закрытие окна [важно; еще не реализовано]

Когда пользователь щелкает на кнопке OK, начинается отсчет времени. Через четыре секунды окно пропадает. Изменения настроек характеристик вступают в силу только по истечении четырех секунд.

3.2.ХИ.4.4. Прерывание [важно; еще не реализовано]

При прерывании окно пропадает.

Заметьте, что прерывания инициируются внешним персонажем, когда он входит в зону. Также заметьте, что в этом случае значения характеристик не изменяются и происходит контакт.

3.3. Требования к производительности

[Примечание для студентов. К требованиям производительности относятся необходимая скорость и (или) временные ограничения. Сюда можно также отнести (статическое или динамическое) использование памяти (RAM, жесткий диск), например данные о памяти, необходимой во время работы программы, если это не было документировано в другом разделе SRS.]

Программа должна загружать и показывать исходную картинку менее чем за минуту.

Контакты должны происходить не дольше одной секунды.

Эти требования были протестированы (здесь должна быть ссылка на тест).

3.4. Ограничения проектирования

[Примечание для студентов. Этот раздел определяет ограничения, накладываемые на проектирование. Если в этом разделе нет материала, разработчики свободны создавать любой (хороший) проект, который удовлетворит остальным требованиям. Например, мы можем добавить ограничение проектирования «одноэтажный» к следующему требованию: «Дом с четырьмя спальнями, каждая из которых не далее тридцати секунд ходьбы из гостиной».]

Проектирование игры *Встреча* должно быть выполнено с использованием UML и объектно-ориентированного подхода. Реализация должна быть выполнена на Java. Программа будет запускаться как Java-приложение Windows 95. Она будет разработана таким образом, чтобы было относительно легко изменять правила игры и другие разработчики имели возможность настраивать игру.

3.5. Атрибуты программной системы

3.5.1. Надежность

Встреча будет зависать не чаще чем один раз из тысячи. Документация тестирования (здесь должна быть ссылка на тест).

3.5.2. Доступность

Встреча должна быть доступна для игры на персональном компьютере под Windows 95 (то есть ни с какой другой программой одновременно). Документация по тестированию (здесь должна быть ссылка).

3.5.3. Защита

[В будущих версиях доступ к сохраненным играм будет разрешен только через пароль.]

3.5.4. Поддержка

3.5.4.1. Изменение персонажей и зон [важно]

Должно быть довольно просто изменять персонажи и зоны.

3.5.4.2. Глобальное изменение стилей [важно]

Должно быть несложно глобально изменить стиль зон и соединений. (Изменения стиля отражают разные уровни игры в одном и том же окружении.)

3.5.4.3. Изменение правил контакта [не обязательно]

Правила контакта должно быть легко изменить.

3.6. Дополнительные требования

Нет.

4. Дополнительная информация

Нет.

4.1. Оглавление и индекс

Будет приложено.

4.2. Приложения

Будут добавлены.

[Примечание для студентов. Приложения могут содержать следующее.

1. *Пример форматов ввода-вывода, описания стоимости изучения и анализа или результаты опросов пользователей.*
2. *Дополнительную информацию, которая может помочь читателям SRS.*
3. *Описание проблемы, которую должна решать программа.*
4. *Специальные инструкции кодирования и способы защиты, экспорта, начальной загрузки и т. д.*

Отдельно уточните, является ли каждое дополнение официальной частью SRS.]

Глава 5
Архитектура программного обеспечения

Как точен и чудесен в действии!
Шекспир. Гамлет

Разработчики программного обеспечения постоянно ищут ясные и изящные архитектуры для своих проектов, поскольку они облегчают получение реализации без ошибок и дефектов, а также лучше приспособлены к расширению и повторному использованию. Содержание этой главы представлено на рис. 5.1.

- Основы: раздел 5.1
- Детали: разделы 5.2–5.6
- Руководство по учебному проекту: архитектура видеоигры *Встреча*
- Упражнения
- Пример: Проектная документация программного обеспечения (SDD) для видеоигры *Встреча*, часть 1

Учебные цели этой главы таковы.

- Понимание термина *архитектура программного обеспечения*.
- Использование каркасов, образцов проектирования и моделей.
- Разработка вариантов архитектуры.
- Отображение архитектуры на детальное проектирование.
- Применение стандарта IEEE SDD.

Основы

5.1. Введение в архитектуру программ

В течение десятилетий разработчики программного обеспечения создавали свои проекты либо с нуля, либо используя уже накопленный опыт, если таковой как-то удавалось приобрести. В настоящее время интенсивно развивается дисциплина

программных архитектур и проектирования. Теперь мы можем говорить об архитектуре высокого уровня и архитектуре низкого уровня в терминах, понятных всем профессиональным разработчикам. Точно так же, как мы восторгаемся такими инженерными разработками, как туннель между Великобританией и Францией или проект Международной космической станции, вскоре мы будем восхищаться грандиозными программными архитектурами.

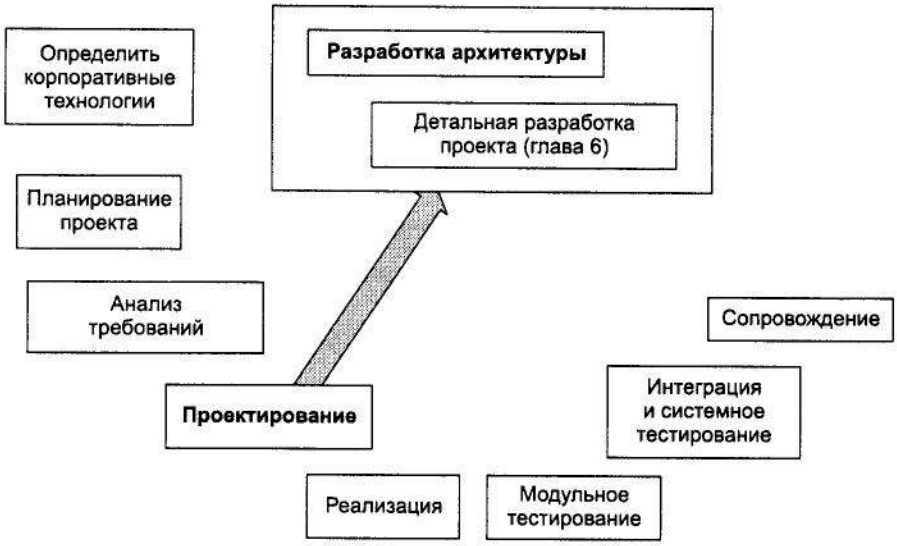


Рис. 5.1. Схема процессов разработки программ: темы главы

5.1.1. Обзор технологии разработки

Любое приложение имеет аппаратные и программные компоненты. Как пример рассмотрим антиблокировочную систему автомобильных тормозов. Она имеет механическую, электронную и программную составляющие и призвана увеличить эффективность торможения, не допуская блокировки колес. Другим примером может служить интерактивный чат в Сети.

Системная разработка — это процесс анализа и проектирования, который разделяет приложение на аппаратные и программные компоненты. Некоторые аспекты этой декомпозиции диктуются требованиями заказчика, другие определяются разработчиками. Рассмотрим процесс декомпозиции на примере возможной конфигурации интернет-версии игры *Встреча*, запускаемой на сервере GameCorp (рис. 5.2).

Процесс системной разработки начинается с определения общих системных требований. Затем делается выбор оптимального соотношения между аппаратным и программным обеспечением. После этого определяется декомпозиция приложения на аппаратное и программное обеспечение. Затем к программной части применяется технология разработки, начиная с анализа требований и т. д. В цели этой книги не входит попытка охватить всевозможные процессы систем-

ной разработки. Мы будем рассматривать проекты с очевидным распределением на аппаратное и программное обеспечение, подобные игре *Встреча*. В данном примере система физически разделена на компьютеры игроков, игровой сервер и сервер обработки счетов (рис. 5.2). Компоненты программного обеспечения распределены между компьютерами так, как показано на рис. 5.2.

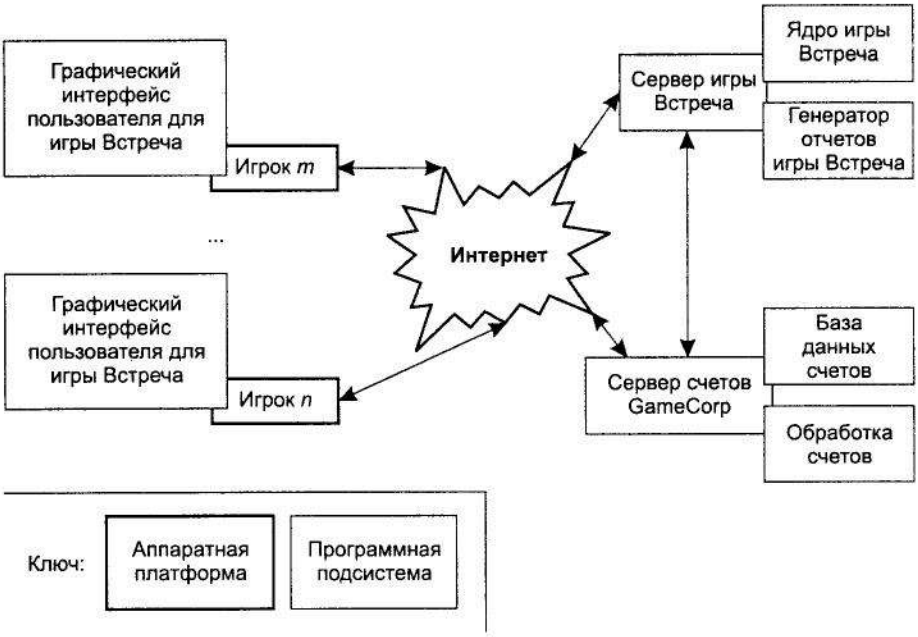


Рис. 5.2. Физическая конфигурация для интернет-ориентированной игры Встреча

Встроенное программное обеспечение взаимодействует на микросекундном уровне с техническими средствами, находящимися в другом месте. Например, программное обеспечение такого рода содержит тормозная система с автоматической антиблокировкой (рис. 5.3). Интернет-ориентированную версию игры *Встреча* (рис. 5.2), напротив, нельзя рассматривать как встроенное программное обеспечение. Встроенные приложения ставят одну из самых сложных задач перед разработчиком, поскольку для них очень большое значение имеет время реакции. Министерство обороны США достаточно давно использует передовые методы системной разработки, поскольку военные системы требуют очень тесной интеграции программного обеспечения и технических средств. Подрядчики министерства обороны используют огромное количество системных разработчиков. Эти разработчики формируют системные требования и проводят исследования, необходимые для создания подходящей конфигурации. Нормы для системных разработчиков определяются в документах IEEE и других стандартах, таких как IEEE P1233. Этот стандарт содержит широкий спектр соглашений по таким вопросам, как размер и вес механических компонентов, ограничения ресурсов, ограничения среды, производительность, функциональность, совместимость, надежность, условия сопровождения и технологичность производства.

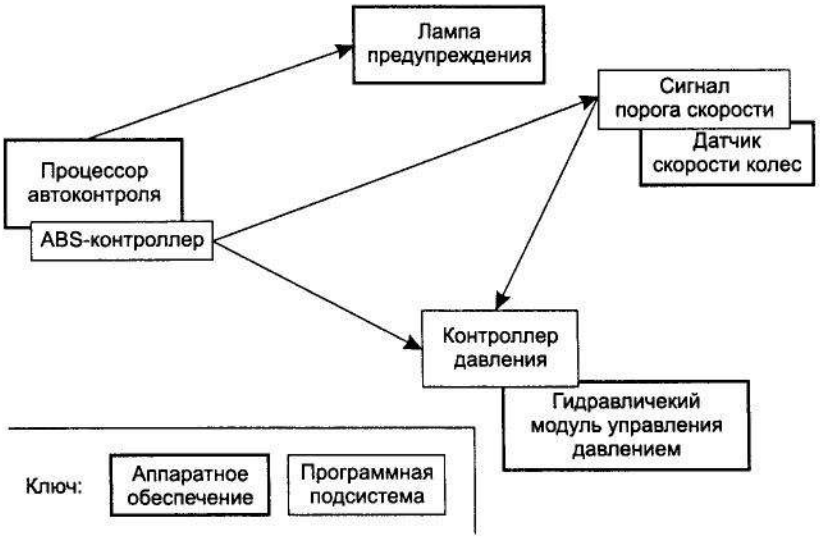


Рис. 5.3. Диаграмма автоматической антиблокировочной системы (ABS) тормозов

5.1.2. Что такое архитектура программы

Если сравнивать разработку программ с процессом постройки моста, то анализ требований будет аналогичен выбору мест, где мост будет начинаться и заканчиваться, а также определению рода нагрузок, которые он должен выдерживать. Далее, архитектор моста должен решить, каким будет мост: подвесным, консольным или какого-то другого типа, удовлетворяющего требованиям. Другими словами, он должен будет определить *архитектуру* моста. Разработчики программного обеспечения сталкиваются с похожим выбором. В этой главе мы рассматриваем выбор архитектуры для приложения.

Создание *архитектуры* — это *проектирование на самом высоком уровне*. Оставшуюся часть процесса проектирования мы будем называть *детальным проектированием*.

Ясное описание архитектуры очень важно для всех приложений и обязательно в том случае, когда к разработке привлекается большое количество людей. Причиной этого служит необходимость разбиения всего приложения на части (модули) с их последующей сборкой. Выбор архитектуры обеспечивает требуемую модульность. Разработчики, которым поручается создание архитектуры (*технические архитекторы*), обычно являются самыми опытными в команде разработки.

5.1.3. Цели выбора архитектуры

Для конкретного проекта разработки программного обеспечения может быть несколько подходящих архитектур, из которых необходимо выбрать лучшую. Обычно бывает сложно удовлетворить все требования, поскольку архитектура может выполнять одно из требований и не выполнять другое. По этой причине

всем требованиям необходимо присвоить приоритеты. Приведем пример списка основных целей разработки.

- Расширение.

  Облегчение добавления новых свойств.
- Изменения.

  Облегчение смены требований.
- Простота:
  - простота понимания;
  - простота реализации.
- Эффективность:
  - достижение высокой скорости: выполнения и (или) компиляции;
  - достижение малого размера: объектного кода и (или) исходного кода.

Расширение определяет степень, в которой архитектура должна поддерживать добавление новых возможностей в приложение. Чаще всего чем лучше архитектура приспособлена к расширению, тем более сложную структуру она имеет и больше времени требуется на разработку. Расширяемость обычно требует введения более высоких абстракций в процесс. Например, мы можем пожелать, чтобы архитектура нашей видеоигры *Встреча* поддерживала не только эту игру (нижний уровень универсальности), но и вообще любую ролевую видеоигру. Универсальность дает множество преимуществ, но ее реализация требует больших затрат времени. Одной из важных задач при выборе степени универсальности является определение класса возможных расширений. Мы не можем проектировать в расчете на все возможные расширения. В связи с этим очень полезны *необязательные* и *желательные* требования, поскольку они показывают, в какую сторону будет направлено развитие приложения.

Разработка с расчетом на изменения преследует другие цели, хотя подразумевает применение тех же приемов проектирования, что и для обеспечения расширяемости. В данном случае мы хотим спроектировать архитектуру таким образом, чтобы она допускала изменение требований, например чтобы требование «игрок должен постоянно иметь полный контроль над своим персонажем» можно было заменить требованием «игрок время от времени случайным образом теряет контроль над своим персонажем».

Простота является целью проектирования при любых обстоятельствах. Простая архитектура, которая допускает расширения и изменения, является редкостью, и ее создание требует больших усилий. К другим критериям, используемым при выборе архитектуры, относятся экономия машинного времени и экономия памяти.

5.1.4. Декомпозиция

После небольшой практики достаточно просто создавать маленькие программы. Большие приложения, однако, ставят перед разработчиками очень трудные задачи, решение которых довольно тяжело достигается на практике. Принципиальная проблема систем программного обеспечения — это их сложность. Сложность

не в смысле количества строк кода, а в смысле их взаимосвязи. Очень хороший способ борьбы со сложностью — разбиение задачи на подзадачи, имеющие характерные свойства небольших программ. По этой причине декомпозиция (или *модуляризация*) является проблемой критической важности и одним из самых интересных этапов разработки. Разработчик первым делом должен представить, как приложение будет работать на высшем уровне, а затем разработать декомпозицию, соответствующую ментальной модели. Например, какие четыре или пять модулей будут реализованы в игре *Встреча*? Или на пять или на шесть модулей разбить персональное финансовое приложение? Следующая проблема, которая встает перед разработчиком, — это декомпозиция уже полученных компонентов, и т. д. Этот процесс иногда называют *рекурсивным проектированием*.

Начнем с целей декомпозиции.

Связность внутри модуля — это сила взаимосвязей между элементами модуля. *Сцепление* характеризует степень взаимодействия модуля с другими модулями. Эффективная модульность достигается максимизацией связности и минимизацией сцепления. Такой способ дает возможность разбивать сложные задачи на более простые. Применение данного подхода к проекту моста иллюстрирует рис. 5.4. Шесть компонентов, полученных при декомпозиции моста, демонстрируют большую степень связности, сцепление же между ними очень мало.

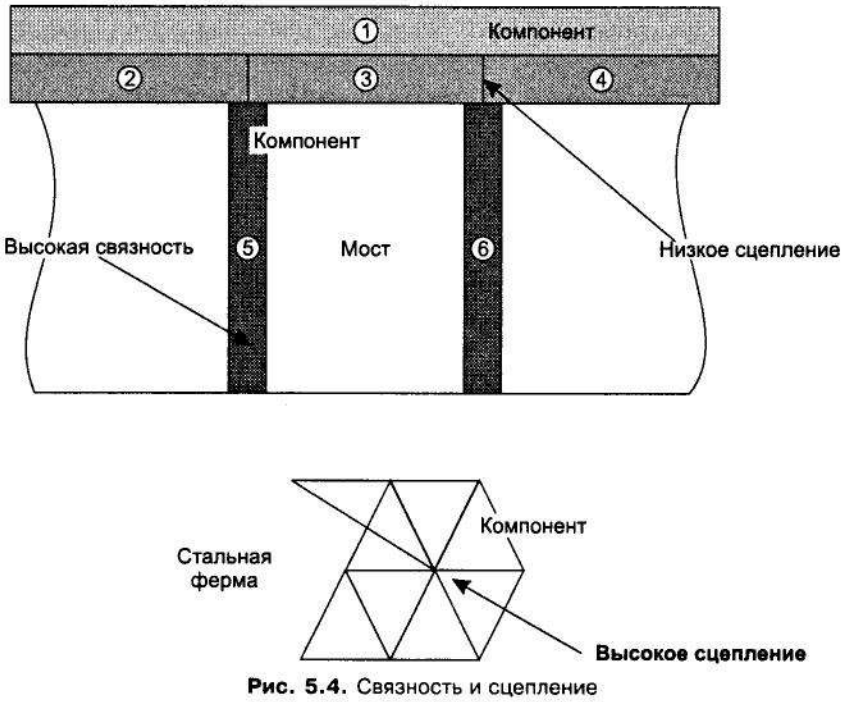


Рис. 5.4. Связность и сцепление

Части (например, кирпичи) каждого компонента моста (например, опоры) взаимозависимы, то есть связность внутри каждого компонента высока. С другой стороны, каждый компонент зависит только от нескольких соседних компонен-

тов, то есть сцепление между ними достаточно низкое. Например, опора связана только с двумя горизонтальными блоками моста. Возьмем другой пример — железную ферму (см. рис. 5.4). Здесь все компоненты имеют одну общую точку, и следовательно, сцепление между этими компонентами очень высоко.

Малое сцепление в совокупности с большой связностью очень важны при проектировании приложений ввиду постоянного процесса внесения изменений в проекты. Сравним жизненный цикл приложения и моста. Изменения в приложении во много раз более вероятны, чем в конструкции моста. Архитектуры с малым сцеплением и большой связностью более приспособлены для модификации, поскольку изменения в таких архитектурах имеют наиболее локальный эффект. Однако создавать такие архитектуры достаточно сложно.

Количество модулей высокого уровня должно быть невелико. Обычно рекомендуемое количество (норматив) — 7±2, но в силу специфики некоторых проектов это число может сильно варьироваться. Разница между мелкомасштабными и крупномасштабными проектами измеряется количеством уровней вложения в модулях. В крупномасштабных приложениях модули высокого уровня разбиваются на подмодули, те в свою очередь разбиваются на под-подмодули и т. д. Норматив 7±2 применяется для каждой такой декомпозиции.

В качестве примера рассмотрим декомпозицию игры *Встреча*. Один вариант — разбить все части игры на четыре модуля.

- Окружающая среда, в которой происходит действие игры (зоны, соединения и т. д.).
- Механизм управления игрой (счетчики, реакции на события и т. д.).
- Участники игры (персонаж игрока, другие персонажи и т. д.).
- Артефакты, задействованные в игре (мечи, книги, щиты и т. д. — они появятся в будущих версиях игры).

Все эти модули обладают достаточной связностью. Например, герои игры достаточно интенсивно взаимодействуют между собой. С другой стороны, сцепление этих модулей сильнее, чем нам бы хотелось. Например, при встрече персонажей задействуются и окружающая среда, и механизмы управления, и сами персонажи, и артефакты.

В качестве другого примера рассмотрим декомпозицию персонального финансового приложения.

- Счета (проверка, сохранение и т. д.).
- Оплата счетов (электронная, чеком и т. д.).
- Глобальные отчеты (общие активы, задолженность и т. д.).
- Займы (автомобиль, образование, дом и т. д.).
- Инвестиции (акции, долговые обязательства и т. д.).

Хотя эта декомпозиция привлекательна с точки зрения пользователя, она имеет большие недостатки как архитектурная декомпозиция. Например, счета имеют малую связность, поскольку они слабо взаимосвязаны. А связность модулей здесь довольно велика. Например, при выплате задолженности задействуются счета, оплата счетов, займы и, возможно, отчеты.

Существует следующая альтернатива этой декомпозиции.

- Интерфейс (пользовательский интерфейс, коммуникационный интерфейс, отчетность и т. д.).
- Поставщики (арендодатель, ссуды, коммунальные услуги и т.д.).
- Активы (проверка счетов, акции, долговые обязательства и т. д.).

Совершенная архитектура — цель достойная, но труднодостижимая. Программирование — это не единственная область инженерии, в которой сложно получить безупречную модульность. В [100] Шнаерсон показал, что, несмотря на все попытки General Motors разбить на модули проект своего первого электрического автомобиля, факторы соответствия форме (требования соответствия деталей автомобиля ограниченному пространству) порождали высокий уровень сцепления между компонентами.

Подведем итоги этого раздела.

ОДИН ИЗ СПОСОБОВ НАЧАТЬ ВЫБОР БАЗОВОЙ АРХИТЕКТУРЫ

1. Разработать ментальную модель приложения на высоком уровне, как если бы это было маленькое приложение. Например, персональное финансовое приложение получает или выдает деньги в любом порядке под управлением интерфейса пользователя.
2. Выполнить декомпозицию на требуемые компоненты. Поиск высокой связности и низкого сцепления. В частности, для персонального финансового приложения выполняется декомпозиция на Активы, Поставщики и Интерфейс.
3. Повторить этот процесс для компонентов.

Список уже известных архитектур приведен далее в этой главе.

Детали

Эта часть главы может быть полностью освоена после прочтения последующих глав. Однако понимание основной идеи этой части необходимо для производства качественных программных продуктов.

5.2. Модели, каркасы и образцы проектирования

Декомпозиция всего проекта на компоненты является существенным шагом, но нам еще предстоит гораздо большая работа по созданию архитектуры. Для начала нам необходимо согласовать варианты использования, классы, переходы состояний и декомпозицию. Мы называем эти проекции *моделями*, и они будут рассмотрены в разделе 5.2.1.

При создании модели классов целесообразно разрабатывать и использовать уже существующее программное обеспечение, которое образует базис для *семейства* сходных приложений. Такое семейство, называемое *каркасом*, рассмотрено в разделе 5.2.3.

Детальное проектирование — это полный объем работ по проектированию, исключая архитектуру и реализацию. Оно содержит определение классов, связывающих классы предметной области и классы архитектуры. Это будет темой обсуждения в главе 6.

Вместо того чтобы «изобретать велосипед», мы стараемся использовать разработки, уже доказавшие свою эффективность в предыдущих приложениях. *Образцы проектирования* — это шаблоны взаимодействующих классов и методов, которые уже показали свое значение для многих приложений. В качестве примера можно взять набор классов, реализующих дерево объектов. Образцы используются как на уровне архитектуры (раздел 5.2.5), так и на уровне детального проектирования (глава 6).

Далее в качестве упражнения мы разработаем каркас классов для игры *Встреча*, общий для всех ролевых игр. Основываясь на этом каркасе, в этой главе мы представим архитектуру игры *Встреча*. Для завершения проектирования данной игры классы предметной области, полученные на этапе анализа требований, необходимо подготовить к работе с классами архитектуры. Мы сделаем это путем добавления классов детального проектирования (глава 6).

5.2.1. Использование моделей

Обычно нам необходимо описать приложение с нескольких точек зрения. Это можно сравнить с архитектурой дома, которая требует нескольких проекций, таких как план расположения участков земли, вертикальный и фронтальный виды, план водопровода и т. д. В мире программирования проекции называются моделями. За последние годы в этой области появилось множество прекрасных разработок. Различные модели проектируемого приложения показаны на рис. 5.5 и 5.6. Многие обозначения взяты из USDP, рассмотренного в [64] и [75].

Рис. 5.5. Модели, рассматривающие приложение с разных точек зрения

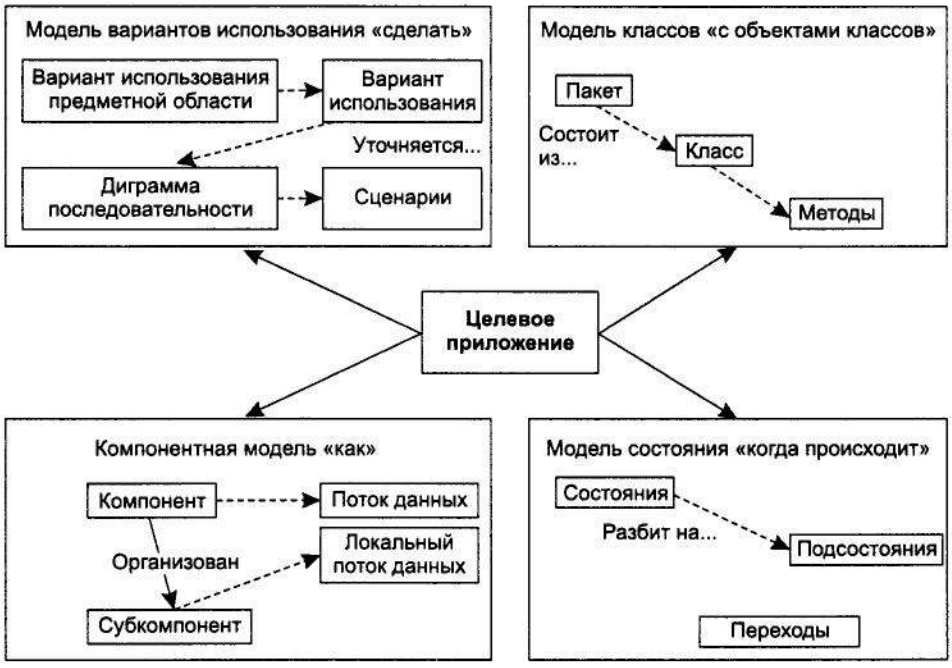


Рис. 5.6. Модели и их части

Модель вариантов использования представляет собой коллекцию вариантов использования. Они поясняют, что именно приложение должно делать. Начальная версия вариантов использования подходит для использования в качестве С-требований (иногда их называют «деловые варианты использования» или «варианты использования предметной области»). В процессе реализации проекта они приобретают свойства диаграмм последовательности специального вида. Их реализуют в виде конкретных сценариев, которые затем используются при тестировании. В качестве примера варианта использования для игры *Встреча* можно взять случай встречи с внешним персонажем, рассмотренный в примере к этой главе.

Модель классов. Мы уже рассмотрели достаточно много моделей классов (диаграмм классов). Модель классов объясняет построение блоков, из которых будет сформировано приложение. Модели классов зачастую называют *объектными моделями*. Внутри модели классов мы можем показать методы и атрибуты.

Модель компонентов является коллекцией диаграмм потоков данных. Они объясняют, каким образом приложение будет работать в терминах перемещения данных. Модель компонентов игры *Встреча* в упражнениях будет включать диаграмму потоков данных о характеристиках между внешним персонажем и персонажем игрока при их встрече.

Модель переходов состояний представляет собой коллекцию диаграмм переходов состояний. Модель переходов состояний определяет момент времени, в который приложение осуществляет свою работу. В игре *Встреча* мы уже разработали такую модель (диаграмма переходов состояний в главе 3). Она показывает реак-

цию игры на появление других персонажей, а также на запросы пользователя установить характеристики, закрыть окно, выйти из игры и т. д.

Внутри каждой модели мы прорабатываем уровни детализации, количество которых растет. В зависимости от объемов работы мы можем итеративно применять деление на уровни детализации внутри каждой из моделей. Архитектура приложения зачастую выражается в терминах одной из моделей и поддерживается остальными моделями. USDP включает в себя модель *реализации*, которая должна быть выполнена с учетом организации кода. В каждой архитектуре есть по меньшей мере одна модель классов, способная реализовать эту архитектуру.

5.2.2. Унифицированный язык моделирования (UML)

В моделях классов, которые будут рассматриваться далее, мы будем широко использовать *унифицированный язык моделирования* — UML (Unified Modeling Language) [15, 95]. Мы уже использовали нотацию UML для вариантов использования и диаграмм последовательности. Основы нотации UML для моделей классов показаны на рис. 5.7 и 5.8. Далее мы рассмотрим эти рисунки подробнее. На самом высоком уровне модели классов для компонентов архитектуры в UML используется термин *пакет*. *Пакеты* — это коллекции классов. *Пакетами* иногда называют коллекции классов Java, и эти термины довольно точно соответствуют друг другу. Пакеты Java транслируются в каталоги, подпакеты — в подкаталоги и т. д. Вообще говоря, UML позволяет помещать в пакеты любые материалы, связанные с приложением, в том числе код, результаты проектирования, документацию и пр.

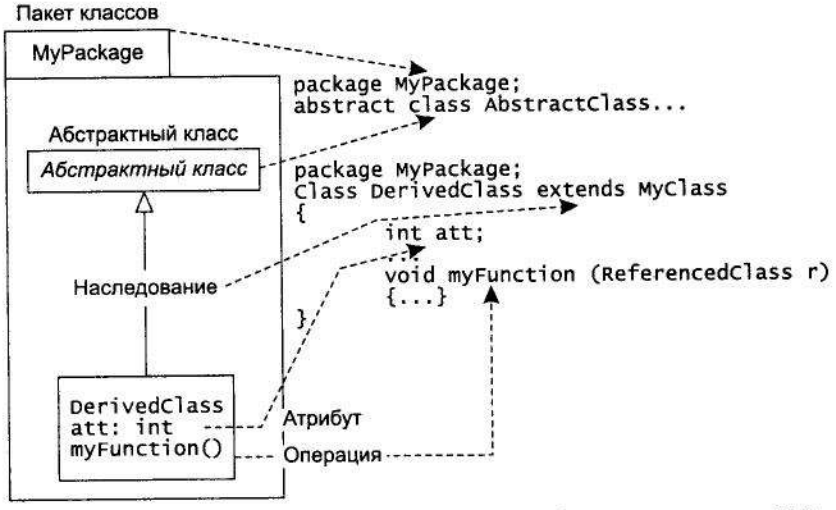


Рис. 5.7. Пакеты, абстракция, наследование, атрибуты и ориентация в UML

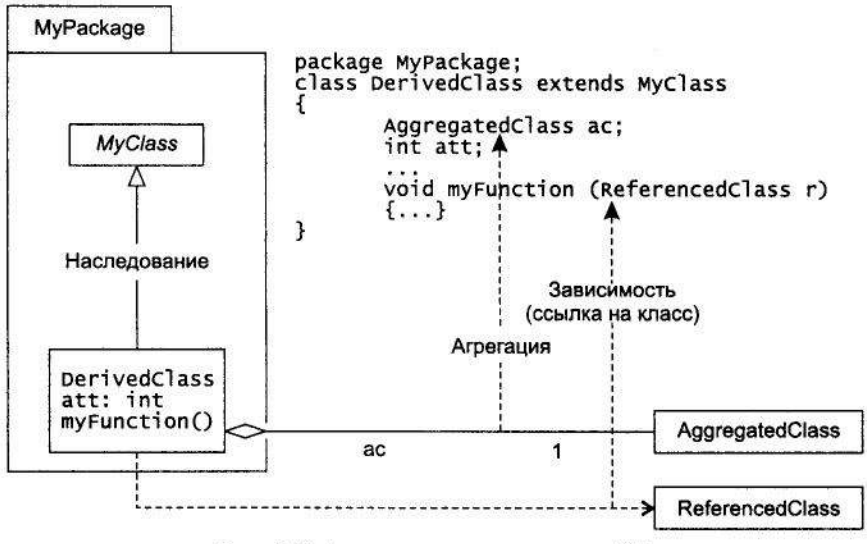


Рис. 5.8. Агрегация и зависимость в UML

Абстрактные классы не имеют экземпляров, то есть объектов (на рисунках имена таких классов выделены курсивом). *Агрегация*, помеченная ромбом, указывает на то, что объекты одного класса являются вложенными в другой класс. Число на полюсе агрегации означает количество агрегированных объектов. Например, число 1 на полюсе отношения *DerivedClass/AggregatedClass* означает, что каждым объектом класса *DerivedClass* агрегируется один объект класса *AggregatedClass*. Вместо числа может быть указан интервал, например 3.7. Символ * означает некоторое число агрегированных объектов. *Зависимость* обозначается стрелкой с пунктирной линией. Обычно она означает ссылку метода зависимого класса на другой класс.

5.2.3. Каркасы

Каркас — это коллекция классов, используемых в нескольких различных приложениях. Часто классы внутри каркаса взаимосвязаны. Они могут быть абстрактными и использоваться через наследование.

Интерфейс прикладного программирования (API), реализованный в Java, является примером полезных каркасных пакетов. Он показал, насколько сообщество разработчиков нуждается в богатой коллекции каркасов для выполнения своей работы. Базовые пакеты Java API могут быть использованы в огромном количестве разнообразных приложений. Связывание пакетов приложений с каркасными пакетами осуществляется при помощи агрегации и (или) наследования. Рассмотрим, например, использование пакета *Java Windowing Toolkit* (*awt*): вместо того чтобы модифицировать *awt*, мы создаем классы графического интерфейса пользователя, которые унаследованы от *awt* или агрегируют объекты *awt* как атрибуты.

Существует мнение, что каркасы следует создавать только в том случае, если они, так же как и Java API, будут использоваться большим количеством приложений. Однако разработка частичного каркаса параллельно с приложением дает множество преимуществ, даже когда нет уверенности в том, что этот каркас будет пригоден для большого числа приложений. Этот частичный каркас нередко служит неизменным абстрактным уровнем, который наследуют классы многих приложений.

В качестве примера создадим каркас, который можно использовать в игре *Встреча*. Выполним декомпозицию игры как приложения. Варианты декомпозиции легко получить, группируя классы предметной области, полученные при анализе требований (*Зона*, *ПерсонажВстречи*, *ИграВстреча*, *Контакт*, *ОкноКонтакта*, *ВнешнийПерсонаж*, *ПерсонажИгрока* и *ОкноХарактеристикИгрока*).

Каждый из этих классов должен подходить для пакета приложения, и каждый пакет приложения должен использовать один или более пакетов каркаса. Например, согласно этому принципу, один каркасный пакет *Персонаж* может определять участие героев в игре *Встреча*. Также может быть создан пакет приложения *ПерсонажВстречи*, использующий *Персонаж* и содержащий классы для персонажей класса *ИграВстреча* (рис. 5.9). Декомпозиция ролевой игры на пакеты каркаса и пакеты приложения *Встреча* показана на рис. 5.10.

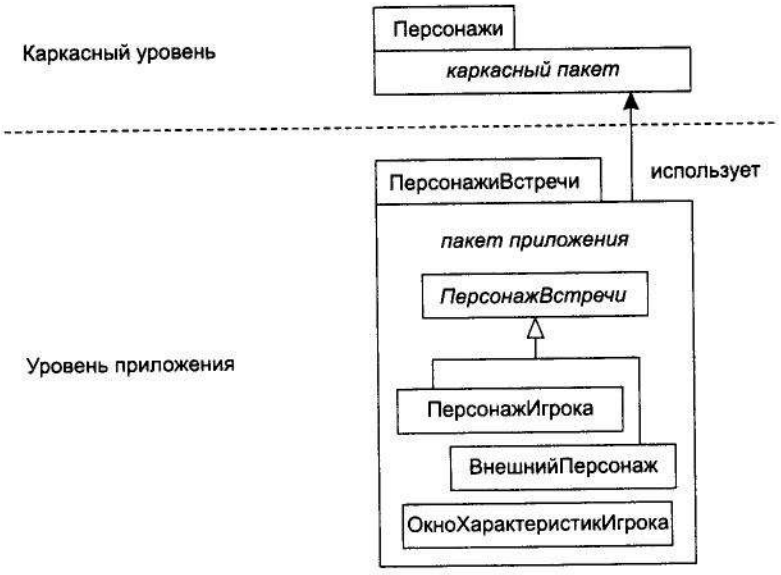


Рис. 5.9. Уровни для ролевых видеоигр

Каркасный пакет *Персонажи* определяет и персонажи, управляемые игроком, и персонажи, управляемые приложением. Каркасный пакет *Артефакты* состоит из элементов, не принадлежащих никаким другим пакетам. Необходимость пакета *АртефактыИгры* на этой стадии еще не совсем ясна. Возможно, еще рано рассуждать о том, должны ли, например, аксессуары рассматриваться как часть уровня или щиты содержаться в пакете *Персонаж*.

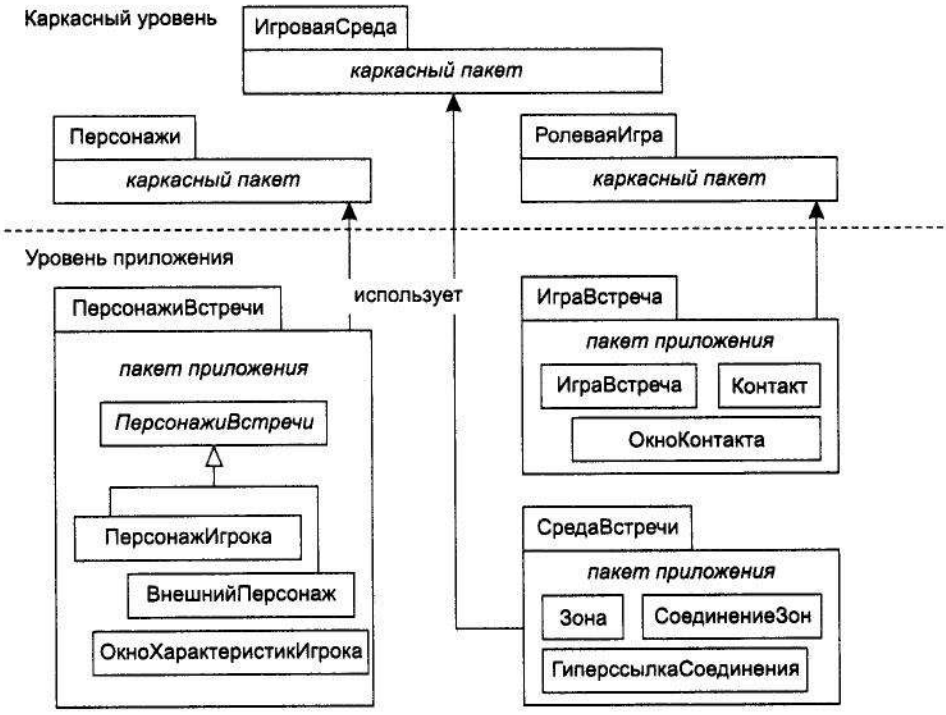


Рис. 5.10. Пакеты архитектуры для ролевых видеоигр, построенные на основе классов предметной области

Мы стремились получить высокую связность и низкое сцепление в приложении *Встреча*, группируя логические сущности (персонажи, игра, управление и отображение). Если бы мы группировали классы так, как они появляются на мониторе (персонажи внутри зоны), то пакеты имели бы неприемлемо низкую связность, поскольку они не являются сильно связанными классами. Сцепление в этом случае оказалась бы очень высоким, поскольку мы должны были бы обеспечить доступ и к зоне, и к персонажу (а это очень разные сущности) извне пакета.

Путь, который приводит нас к модели классов, представлен на рис. 5.11. Классы предметной области получены как результат анализа требований. Классы каркаса получены из разработки архитектуры для приложений, подобных тому, которое мы проектируем. Таким образом, классы каркаса либо являются частью ранее существовавшего пакета, либо разработаны, как в процессе анализа архитектуры. Наконец, оставшиеся классы (*проектные* классы) добавлены для завершения проектирования. Завершенное проектирование состоит из всех классов предметной области, всех классов проектирования и некоторых каркасных классов (рис. 5.12). (Обычно мы не используем все каркасные классы.) Каркасные классы, использованные в проекте, являются частью архитектуры приложения. Все классы предметной области — это часть детального проектирования, поскольку

они специфичны для каждого приложения, получены из требований и не относятся к архитектуре. В качестве примера возьмем класс *ПерсонажИгрока* из упражнения. По сути, он является отдельным классом, описывающим поведение персонажа. Некоторые из проектных классов создаются в процессе проектирования архитектуры, другие — в процессе детального проектирования.

Рис. 5.11. Один из способов получения модели классов

Рис. 5.12. Сравнение модели классов с архитектурой и детальным проектированием

5.2.4. Классификация архитектур

Шоу и Гарлан [34] классифицировали архитектуры программного обеспечения с точки зрения практики. Другими словами, они собрали вместе образцы программного обеспечения для различных архитектур. Их классификация, немного адаптированная, показана ниже.

- Архитектуры потоков данных.
  - Последовательные пакеты.
  - Каналы и фильтры.

- Независимые компоненты.
  - Параллельные взаимодействующие процессы.
  - Клиент-серверные системы.
  - Системы, управляемые событиями.
- Виртуальные машины.
  - Интерпретаторы.
  - Системы, основанные на правилах.
- Репозиторные архитектуры.
  - Базы данных.
  - Гипертекстовые системы.
  - Доски объявлений.
- Уровневые арихитектуры.

Большинство из этих архитектур подробно рассматриваются в разделе 5.3. Существует широкий спектр проблем, требующих программного решения, но существует также широкий спектр архитектур, необходимых для их решения. Возможно, одна из архитектур, определенных Шоу и Гарланом, будет соответствовать вашей задаче или хотя бы подскажет идею декомпозиции.

5.2.5. Образцы проектирования I: введение

Образец проектирования — это найденная опытным путем комбинация компонентов, обычно классов или объектов, которая решает определенные общие проектировочные задачи. Воспользуемся аналогией с архитектурой дома и рассмотрим задачу проектирования уединенного здания на обширной территории. Архитектура *Ранчо* (одноэтажный дом) полностью удовлетворяет этим требованиям. Заметьте, что *Ранчо* указывает на общую идею проектирования, предусматривающую множество реализаций, и совсем не является неизменным множеством планов дома.

Гамма представил вниманию сообщества разработчиков образцы проектирования в теперь уже классической книге [33]. Гамма рассматривает двадцать три образца проектирования, разделяя их на *структурную*, *креационную* и *поведенческую* категории. Структурные образцы проектирования имеют дело со способами представления объектов (такими, как деревья или связные списки). Они удобны во многих случаях, поскольку позволяют пользоваться множеством объектов как единым целым. Креационный образец проектирования связан со способами создания сложных объектов, таких как лабиринты и деревья. Поведенческий образец проектирования позволяет нам следить за поведением объектов, например, выдавая отчет о коллекции объектов в определенном порядке. Хотя существует гораздо больше образцов проектирования, чем приведено в [33], мы все же сосредоточим наше внимание на применении рассмотренных образцов проектирования.

Образцы проектирования могут быть применены на уровне архитектуры и (или) на уровне детального проектирования. Список образцов проектирования, особенно полезных на уровне архитектуры, представлен в табл. 5.1. Далее в этой главе мы постепенно раскроем их смысл.

Таблица 5.1. Обобщение образцов проектирования архитектуры

Цели проектирования	Образец проектирования	См. раздел
Обеспечить интерфейс для множества объектов различных классов	Facade	5.3.2.1
Заставить объект вести себя соответственно его текущему состоянию	State	5.3.2.3.
Инкапсулировать пути «посещения» объектов в коллекции так, чтобы клиентский код мог выбирать пути «посещения» во время выполнения	Iterator	5.3.4.1
Обеспечить реакцию нужных элементов на изменения в источнике данных	Observer	5.3.2.2.1
Интерпретировать выражения на языке формальной грамматики	Interpreter	5.3.4.1

Применение каждого образца проектирования зависит от клиента — кода, который нуждается в сервисе, предоставляемом образцом проектирования. Клиент ссылается на точку входа образца проектирования (обычно это метод класса внутри образца). Кроме того, обычно подразумевается и третий тип кода, который можно назвать *установочным кодом*. Он устанавливает состояние образца проектирования. Установочный код не предназначен для повторного использования и предоставляет клиентам возможность легко взаимодействовать с образцом проектирования. В частности, для работы с образцом проектирования клиенту следует знать как можно меньше о его структуре и внутренней работе.

5.2.6. Компоненты

Во второй половине 90-х годов сильно возрос интерес к понятию «компонент». *Компонентами* являются повторно используемые объекты, которые не требуют знания программного обеспечения, использующего их. Наглядным примером технологии компонентов могут служить объекты COM и Java Beans. Компоненты могут быть объектами в обычном понимании объектно-ориентированного программирования, но с небольшими поправками на обеспечение их автономности. Одни компоненты используются другими с помощью агрегирования и взаимодействуют в основном с помощью событий.

Обобщение связей между каркасом, архитектурой, детальным проектированием, моделями и образцами проектирования показано на рис. 5.13. Образцы проектирования могут быть использованы и на уровне каркаса, и внутри проектных классов на уровне архитектуры, и на уровне детального проектирования.

Обычно никто не пытается применить образцы проектирования внутри классов предметной области, поскольку последние разработаны индивидуально и соответствуют непосредственно множеству требований. Как далее будет показано, образцы проектирования зачастую требуют введения классов, не являющихся классами предметной области, например абстрактных классов.

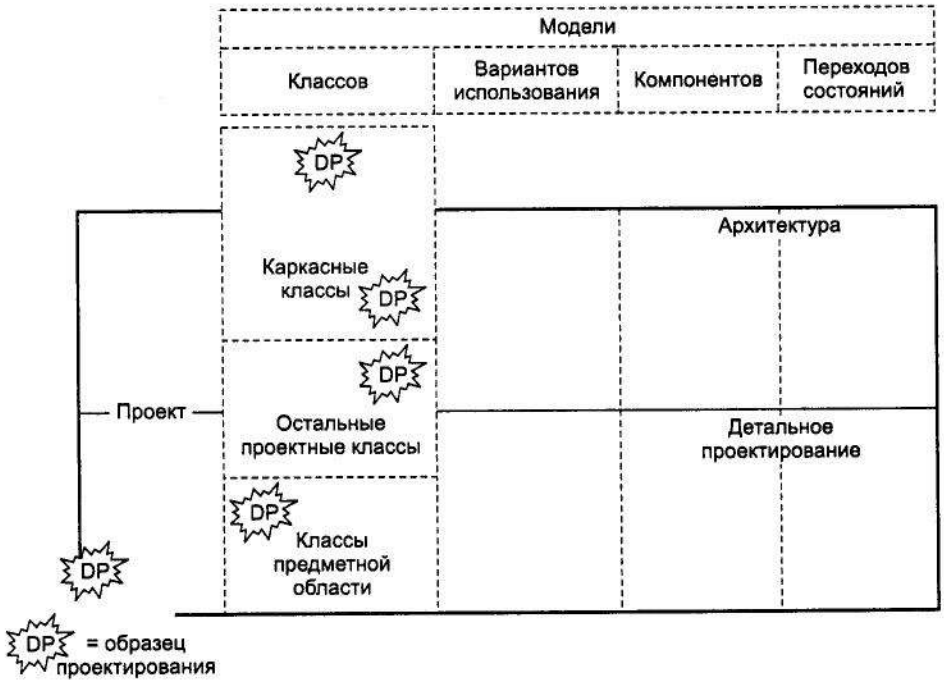


Рис. 5.13. Связи между каркасами, архитектурой, проектированием и моделями

5.3. Типы архитектур и их модели

Разработчик программного обеспечения создает ментальную модель, описывающую работу приложения, ограничиваясь при этом пятью-семью компонентами (весьма приблизительная оценка). Результат проектирования, конечно, определяется в основном приложением, однако многое может быть позаимствовано из ранее разработанных архитектур, так же как в проект подвесного моста войдет множество преимуществ ранее разработанных подвесных мостов. В этом разделе мы подробно рассмотрим архитектуры, классифицированные в [34], и укажем образцы проектирования, которые могут помочь в реализации этих архитектур.

Сначала мы обобщим встречающиеся типы архитектур в табл. 5.2, а затем поясним большинство из них.

Таблица 5.2. Типы архитектур (по классификации Гарлана и Шоу)

Категория	Подкатегория	Часто применяемые образцы проектирования	Примечания
Поток данных	Последовательность пакетов	Может применяться образец Decorator ([33])	
Независимые компоненты	Каналы и фильтры		
	Параллельные взаимодействующие процессы	Observer (раздел 5.3.2.2.1)	
	Клиент-серверные системы	Facade (раздел 5.3.2.1)	
	Системы, управляемые событиями	State (раздел 5.3.2.3), Observer	
Виртуальные машины	Интерпретаторы	Interpreter (раздел 5.3.3)	
	Системы на основе правил		Объяснения правил в [43]
Репозиторные архитектуры	Базы данных	Observer, Iterator (раздел 5.3.4.1)	
	Гипертекстовые системы		См. Decorator в [33]
	Доски объявлений		Определение досок объявлений в [29]
Уровневые архитектуры			Большинство образцов проектирования состоят из абстрактного и неабстрактного уровней

5.3.1. Архитектуры, основанные на потоках данных

Для представления некоторых приложений наилучшим образом подходят потоки данных между процессами обработки данных. Такое представление иллюстрируют *диаграммы потоков данных* (DFD — Data Flow Diagram). Каждый процесс обработки данных на диаграмме потоков данных проектируется независимо от других. Данные приходят из различных источников, например от пользователя, и, в конечном итоге, возвращаются к пользователю или в приемники данных, такие как база данных счетов. Элементы диаграммы потоков данных были рассмотрены в разделе 3.3.3. Диаграмма потоков данных для банковского приложения представлена на рис. 5.14.

Данные передаются от пользователя к процессу *Получение депозита*. Далее процесс *Получение депозита* посылает номер счета и размер депозита процессу, проверяющему данные на совместимость. Если данные корректны, то они могут

быть отправлены процессу, создающему транзакцию, и т. д. Например, процесс *Создание транзакции запроса* сам по себе может быть разбит на более подробные диаграммы потоков данных. Эта тема будет рассмотрена в главе 6.

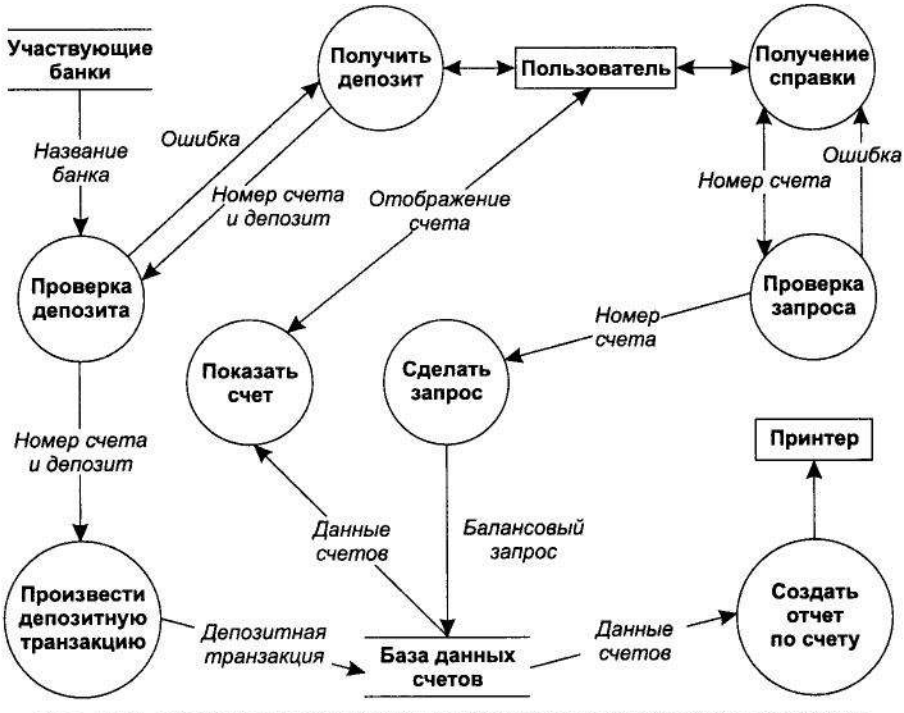


Рис. 5.14. Частичная диаграмма потоков данных для банковского приложения

Архитектура потоков данных, относящаяся к архитектурам типа *каналы и фильтры*, представлена на рис. 5.15. Такие архитектуры потоков данных состоят из процессов (*фильтров*), способных в любой момент времени принять *потоки* как входную информацию (последовательность данных единообразного вида). Каждый фильтр должен проектироваться независимо от других. Такая архитектура может быть легко реализована с помощью каналов Unix.

Архитектура каналов и фильтров имеет явное преимущество — модульность. Пример приложения с такой архитектурой показан на рис. 5.16. Это приложение обслуживает транзакции счетов, приходящие в случайные моменты времени по линиям связи. Архитектура содержит шаг для регистрации транзакции в случае отказа системы. Функция *снятие()* получает в качестве входного параметра символьную строку вида `ДжонДоуНомерСчета12345Суммат$3500.00` или `ДжонДоуНомерСчета12345Суммат$3500.00`, а также банковский адрес вида `НомерБанка9876`. Процессы, изображенные в эллипсах, ожидают появления значения на входе, для того чтобы начать свою работу.

В общем случае не существует унифицированного способа изображения диаграммы потоков данных для моделей классов. Однако функциональные модули

диаграммы потоков данных могут иногда указывать прямо на методы классов. Примером является диаграмма на рис. 5.16.

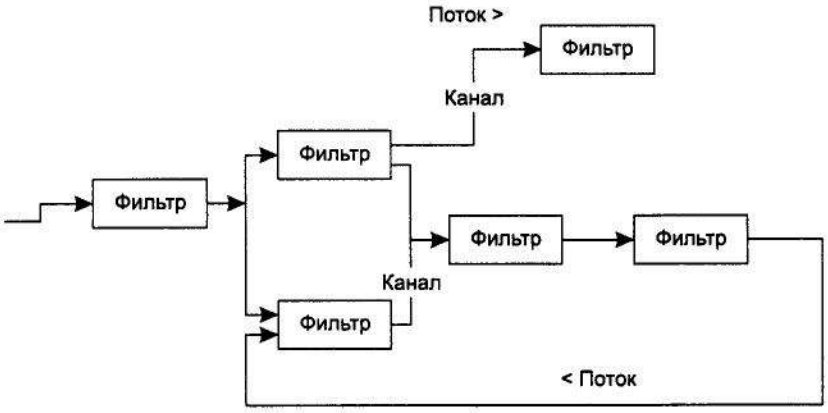


Рис. 5.15. Архитектура каналов и фильтров

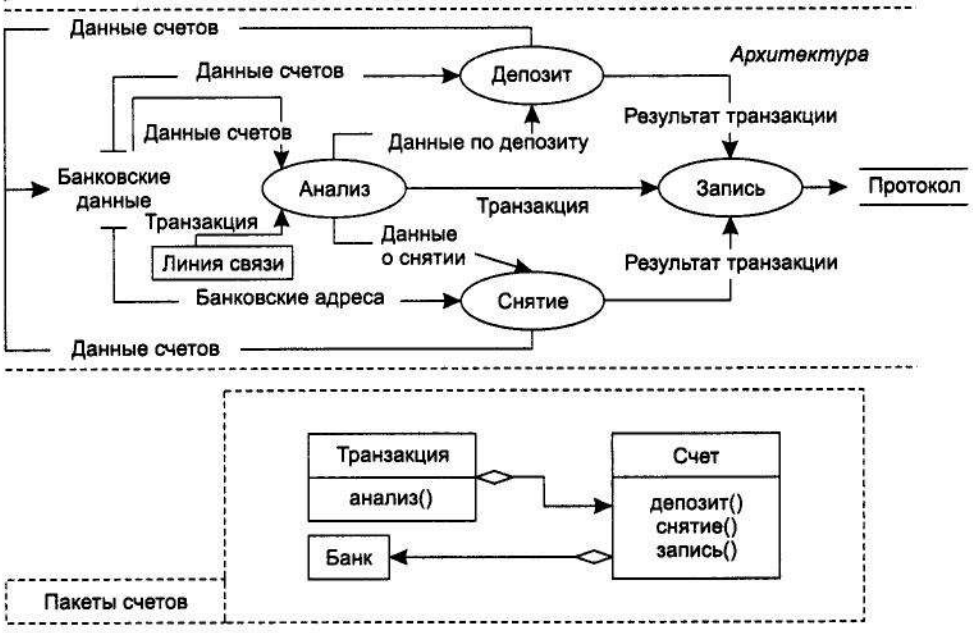


Рис. 5.16. Пример выбора потоков данных в архитектуре каналов и фильтров

Благодаря росту объемов распределенных вычислений увеличивается количество приложений с потоко-ориентированными вычислениями. Причиной является тот факт, что передача данных зачастую осуществляется в виде форматированных символьных строк. Такой способ передачи данных реализован, например, в удаленном вызове метода (RMI — Remote Method Invocation) в Java. RMI пре-

образует передаваемые объекты в символьные строки. Кроме того, часто ввод-вывод также бывает реализован с применением потоков, поэтому использование ввода-вывода в таких языках, как Java, как правило, аналогично процессам-фильтрам.

В частном случае, когда на вход фильтров подается информация только в виде пакетов, результатом будет поток данных в виде последовательности пакетов. В качестве примера рассмотрим банковское приложение, вычисляющее, сколько денег свободно для выдачи ссуд под заклад и для выдачи необеспеченных ссуд. Диаграмма потоков данных этого приложения представлена на рис. 5.17. Эта диаграмма пакетно-последовательная, поскольку функции используют для своей работы фактически всю поступающую информацию. Например, подсчитывая фонды для ссуд под заклад, функции используют информацию практически обо всех счетах. Этот пример существенно отличается от предыдущего примера транзакции (см. рис. 5.16), поскольку там мы имели множество (потенциально неограниченное) транзакций, использующих выделенные данные из своих источников.

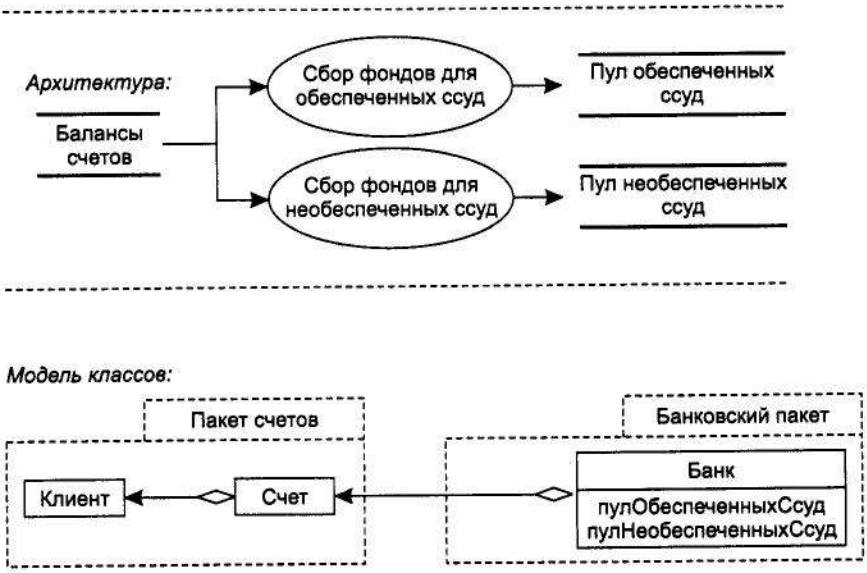


Рис. 5.17. Пример архитектуры пакетного последовательного потока данных

Рисунок 5.17 также иллюстрирует одно отображение на классовую модель, в которой функции диаграммы потоков данных реализованы в виде методов класса *Банк*. *Пакеты* выполняются путем вызова соответствующих методов этого класса.

В течение десятилетий потоки данных являлись наиболее общим способом отображения архитектур, и можно с уверенностью сказать, что они не потеряют актуальности и в будущем. Для разработчиков вполне естественно представлять себе, что данные путешествуют от одного узла к другому и обрабатываются в каждом узле. Недостатком диаграмм потоков данных является то, что их отображение на программный код не вполне ясно, причем независимо от того, идет ли речь об объектно-ориентированном коде, или нет.

Мы воспользуемся моделью потоков данных еще раз при рассмотрении детального проектирования в следующей главе.

5.3.2. Независимые компоненты

Архитектура *независимых компонентов* состоит из компонентов, работающих параллельно (по крайней мере, теоретически) и время от времени общающихся друг с другом. Возможно, самый очевидный пример можно найти в World Wide Web, где тысячи серверов и миллионы браузеров все время работают параллельно и иногда общаются между собой.

5.3.2.1. Клиент-серверная архитектура и образец проектирования Facade

В отношениях клиент-сервер серверные компоненты обслуживают нужды клиентов с помощью запросов. Клиент-серверные отношения имеют огромное преимущество — малое сцепление между компонентами. Такие отношения применяются в разработке программного обеспечения в основном в тех случаях, когда в реализации участвует более одного человека: вполне естественно разделить пакеты классов на части для каждого разработчика из группы. При этом разработчики обычно используют сервисы классов, за которые отвечают другие разработчики. Другими словами, пакеты разработчиков часто связаны такими же отношениями, что и серверы с клиентами. Обычно для этой ситуации характерна одна проблема — различные уровни готовности компонентов в процессе разработки.

Компонент работает как сервер более эффективно в случае, если он имеет узкий интерфейс. «Узкий» означает, что интерфейс (по сути, набор функций) не имеет излишеств, собран в одном месте и четко определен. Образец проектирования *Facade* предоставляет пакетам классов именно такой интерфейс. Facade регулирует взаимодействие с другими объектами, предоставляя для использования только один объект и скрывая остальные. Этот открытый объект обычно единственный и является членом класса `Facade`. Структура образца проектирования Facade изображена на рис. 5.18.

Вызов, ссылающийся на объект внутри пакета, преобразуется в вызов метода объекта `Facade`. Этот метод и ссылается на объект, требуемый вызовом.

Использование образца проектирования Facade для каждого пакета игры *Встреча* показано на рис. 5.19. Взаимодействие с персонажами игры должно осуществляться через одиночный объект класса *РолиВстречи*. Ссылки на элементы среды игры *Встреча* должны осуществляться через объект *СредаВстречи*.

Одна из трудностей, возникающих при применении образца проектирования Facade, связана с запретом на обращение к внутренним классам пакета. Например, пользователям пакета *ПерсонажиВстречи* обычно необходим доступ к отдельным персонажам игры, но они имеют доступ только к классу *РолиВстречи*. Более того, пользователи пакета *ПерсонажиВстречи* не могут даже ссылаться на класс *ПерсонажВстречи*, поскольку это противоречило бы принципу сокрытия

данного класса. Это строгое ограничение. Однако если пакет каркаса (открытый) содержит класс *ПерсонажИгры*, то пользователи пакета *ПерсонажиВстречи* могут ссылаться на класс *ПерсонажИгры*. Обычно для управления доступом этого достаточно.

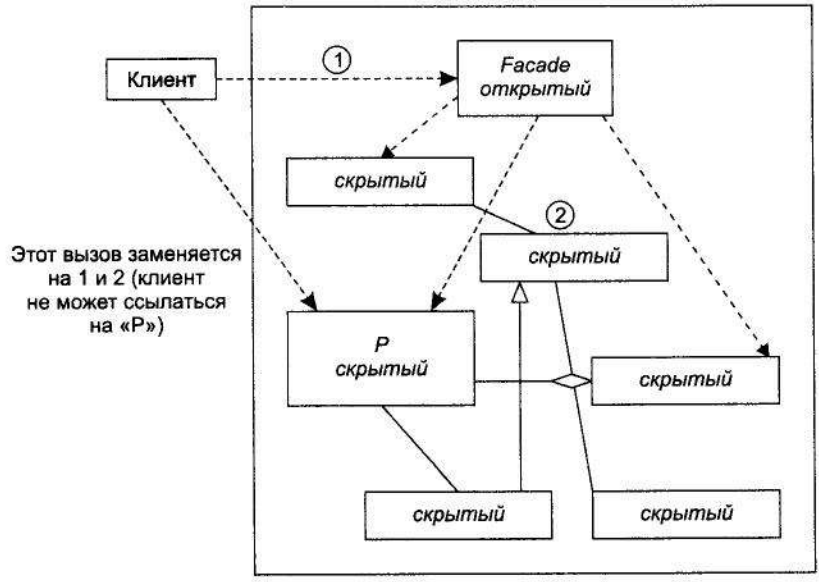


Рис. 5.18. Структура образца проектирования Facade

Рис. 5.19. Архитектура и декомпозиция игры Встреча

В частности, facade-класс пакета *ПерсонажиВстречи* может иметь, например, такие открытые методы:

```
ПерсонажИгры getMainCharacter();
```

В этом случае клиентский код может выглядеть так:

```
ПерсонажИгры главныйПерсонажИгрока = ПерсонажиВстречи.getMainCharacter();
```

Все разрешенные действия, влияющие на главный персонаж игрока, управляются facade-объектом *ПерсонажиВстречи*.

Можно провести простую аналогию с подобным использованием образца проектирования Facade. Предположим, что вы звоните в фирму CampaLot Corp. с намерением купить палатку. Для этого вам необходимо знать о палатках хотя бы в общих чертах (это похоже на возможность ссылаться на каркасный класс *Палатка*). Но, кроме того, вы рассчитываете на интерфейс CampaLot Corp. (человека на другом конце телефонного провода — объект Facade), который обеспечит вам доступ к конкретному виду палаток.

Рассматривая применение образца проектирования Facade, следует учитывать дополнительные затраты на обеспечение доступа пользователей пакета к методам. Например, предположим, что у нас есть пакет `Chemistry`. Он содержит класс `Molecule`, который в свою очередь содержит полезный метод `atomicWeight()`. Если мы используем facade-класс `MoleculeFacade`, то `MoleculeFacade` должен содержать открытый метод `getAtomicWeight()`, передающий управление методу `atomicWeight()`. Это требует дополнительной работы и согласования во время процесса разработки, но затраты на это с лихвой окупятся при возрастании модульности.

Клиент-серверные архитектуры были широко распространены в 80-х и 90-х годах прошлого века. Многие из них пришли на смену архитектурам «центральная машина — терминалы». Клиент-серверные архитектуры усложнились, стали более разнообразными. Некоторые из них являются трехуровневыми, в отличие от классических двухуровневых (клиент и сервер). Третий уровень находится между клиентом и сервером и отвечает за перенаправление данных и их преобразование. Обычно распределение по уровням таково: пользовательский интерфейс разрабатывается для клиентского уровня; управление процедурами — для среднего уровня; сами базы данных — для третьего. Средний уровень может быть общей *шиной* данных, примером которой является стандарт CORBA (Common Object Request Broker — Общая архитектура посредника запросов к объектам). Возможен альтернативный вариант, когда средний уровень функционирует на основе бинарного стандарта, например COM. Наконец, World Wide Web можно рассматривать как целое поколение клиент-серверных архитектур, в котором архитектуру один сервер — десятки клиентов заменила архитектура один сервер — миллионы клиентов.

5.3.2.2. Архитектура параллельных взаимодействующих процессоров

Еще один тип архитектур *независимых компонентов* был определен Шоу и Гарланом как архитектура *параллельных взаимодействующих процессов*. Такая архитектура характеризуется тем, что в ней одновременно запускаются несколько процессов (в разных потоках, выражаясь в терминах Java). Элементы этой архитектуры используются в игре *Встреча*: внешний персонаж Фредди перемещается независимо из одной зоны в другую параллельно с выполнением программы самой игры. Его поток «общается» с другими потоками, когда бы Фредди ни оказался в одной зоне с персонажем игрока.

Нотация UML, выражающая параллелизм, была рассмотрена в главе 4. С помощью этой нотации мы представили архитектуру банковского приложения (рис. 5.20), которое предназначено для обработки множественных транзакций, появляющихся последовательно на банкомате.

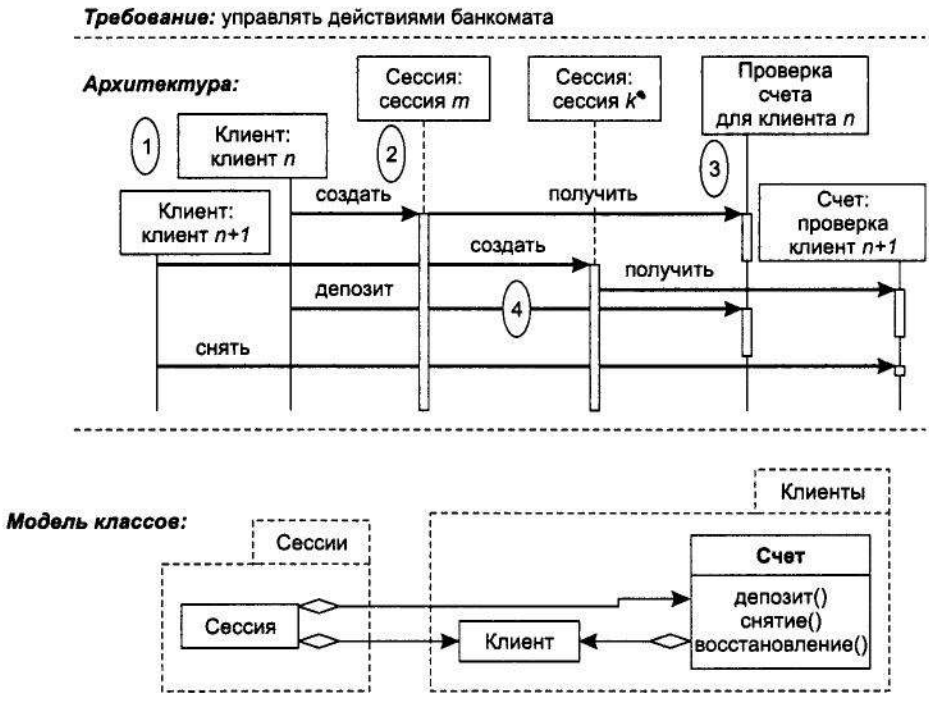


Рис. 5.20. Пример архитектуры параллельных взаимодействующих процессов

Когда клиент *n* пользуется банкоматом, создается объект *клиент n* (1 на рис. 5.20). Этот объект создает поток или параллельный процесс *сессия m* (2), что на диаграмме отмечается половинной стрелкой. Процесс *сессия m* извлекает объект класса *Счет*, например *проверка клиента n* (3). Затем клиент выполняет депозитную транзакцию над проверяющим объектом (4). Параллельно с этим создаются другие объекты *Клиент*, такие как *клиент n+1*, и исполняются в других потоках, например в *сессии k*.

Если в приложении необходимы параллельные процессы, наилучшим выбором является архитектура *параллельных взаимодействующих процессов*. Эта архитектура может быть использована для схем, координирующих концептуально независимые задачи. В классической книге [25] Дийкстра показал, что рассмотрение процесса как комбинации параллельных составляющих может зачастую упростить разработку. Примером тому является имитация клиентов банка. Традиционно такие имитации разрабатывались без параллелизма — они просто сохраняли и обрабатывали события. Однако проектирование в подобных случаях можно упростить, если рассматривать действия каждого клиента независимо. Такая архитектура параллельно взаимодействующих процессов имеет существенное преимущество — она более всего соответствует действиям, которые имитируются.

Более подробно параллельные взаимодействующие процессы в контексте Java рассмотрены в [76].

5.3.2.2.1. Образец проектирования Observer

Архитектура *независимых элементов* зачастую складывается из источника данных и некоторого количества клиентов. Данные клиентов должны обновляться каждый раз, когда в источнике происходит какое-либо изменение. Например, предположим, что штаб-квартира Международной корпорации гамбургеров хранит на своем сервере данные о продаже гамбургеров по всему миру. Распределенными клиентами для этих данных являются Главное управление компании, отдел маркетинга и отдел продаж. Данные постоянно изменяются, и каждому из клиентов необходимо обновлять отображение данных в соответствии с его требованиями и запросами. Например, гистограмма в Главном управлении компании обновляется всякий раз, когда происходит 5-процентное изменение, для отдела маркетинга отображается новая круговая диаграмма, когда происходит изменение по крайней мере на 1 %, а в отделе продаж данные, отображаемые в таблицах, обновляются при любом изменении.

В качестве архитектуры, способной удовлетворять этим требованиям, можно использовать образец проектирования *Observer*. Стороны, которым требуется обновлять данные, называются *наблюдателями* и являются подклассами одного абстрактного класса, который мы назовем `Observer`. Схема образца проектирования Observer изображена на рис. 5.21. Проследим за порядком работы класса `Observer`.

1. Клиент ссылается на фиксированный объект, требуя извещения для наблюдателей. Клиентский объект может представлять собой процесс, запрограммированный отмечать изменения данных, или работающее по расписанию задание. В такой модели, как мы видим, объект-клиент сообщает объекту-источнику, чтобы тот выполнил свою функцию извещения `notify()`.
2. Метод извещения `notify()` сообщает всем объектам-наблюдателям, что им необходимо выполнить функцию обновления `update()`.
3. Способ выполнения метода `update()` зависит от конкретного наблюдателя (`ConcreteObserver`), которому он принадлежит. Функция `update()` сравнивает состояние объекта `ConcreteObserver` (по значениям переменных) с состоянием центрального источника данных на сервере, а затем решает, менять ли значения своих переменных, чтобы привести их в соответствие с источником данных, или выполнить другие действия, например создать новое отображение.

Применив схему образца проектирования Observer для решения задачи с Международной корпорацией гамбургеров, мы получаем архитектуру, изображенную на рис. 5.22. Преимуществом этого образца проектирования является то, что он получил широкое признание (в языке Java, например, существуют даже такие классы: `Observer` (Наблюдатель) и `Observable` (Наблюдаемое)). Другое преимущество состоит в том, что такая модель позволяет ликвидировать наблюдатели и создавать новые, не нарушая взаимодействия остальных наблюдателей. Недостаток образца проектирования Observer проявляется в том случае,

если очень немногим из наблюдателей необходимо реагировать на происходящие изменения: в таком случае многочисленные извещения становятся причиной бесполезной траты ресурсов. Кроме того, эту модель также нежелательно применять, когда политика обновления предлагается и определяется более естественно самими наблюдателями или когда она значительно отличается для разных наблюдателей.

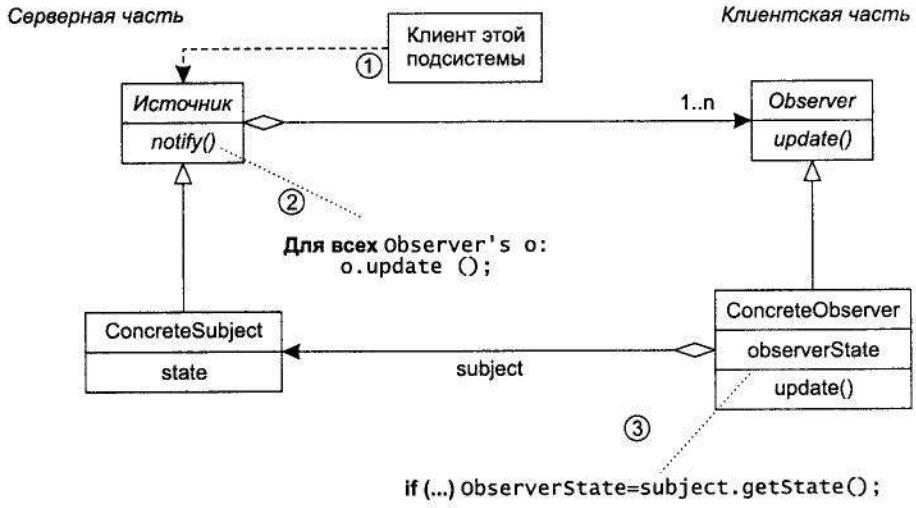


Рис. 5.21. Образец проектирования Observer

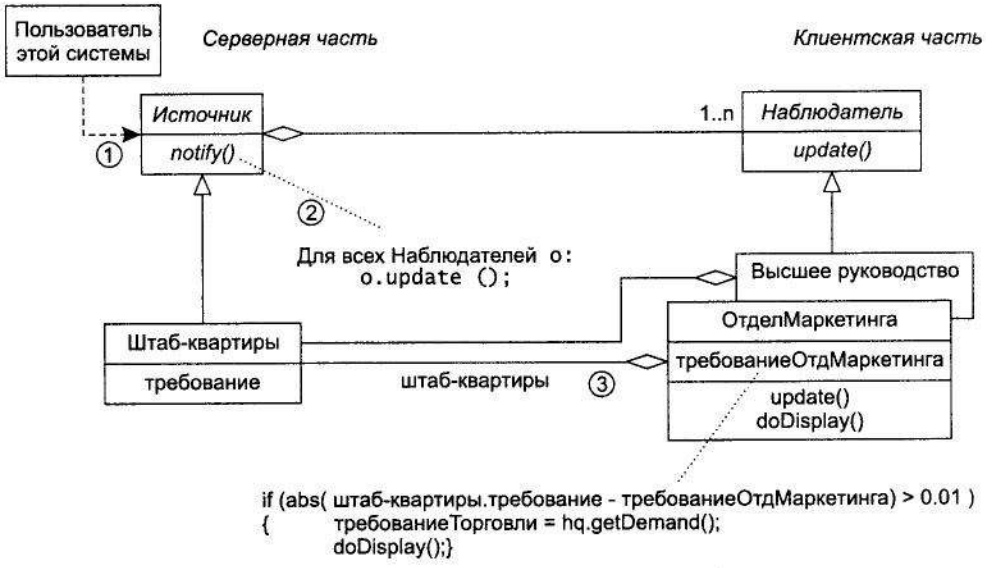


Рис. 5.22. Применение образца проектирования Observer для Международной корпорации гамбургеров

5.3.2.3. Архитектуры событийно-управляемых систем и образец проектирования State

Рассмотрим *событийно-управляемые системы* — третий тип архитектуры *независимых компонентов*, предложенной Шоу и Гарланом. В данной архитектуре приложения представляются состоящими из набора компонентов, каждый из которых находится в состоянии ожидания, пока не произойдет воздействующее на него событие. К таким системам относятся многие современные приложения. Например, текстовый редактор Word ждет, пока пользователь не щелкнет на значке или на пункте меню. Далее происходит соответствующая реакция: сохранение файла, изменение шрифта и т. п. Системы, управляемые событиями, часто строятся на основе модели переходов состояний (см. главу 3).

Если по сути поведение системы заключается в смене *состояний* и задан набор *состояний*, через которые она переходит, то при проектировании подобной системы следует подумать об использовании образца проектирования *State*. Например, общее требование для игры *Встреча* мы описали в виде диаграммы переходов состояний (см. рис. 3.11). Игра *Встреча* может находиться в таких состояниях, как *Настройка*, *Ожидание*, *Установка характеристик*, *Оповещение*, *Контакт*, а также некоторых других. Наша модель проектирования эффективно отражает данное поведение. Эта модель также должна быть способна усваивать новые состояния и уметь управлять действиями, когда проектирование игры подходит к стадии завершения; при этом не должна нарушаться уже существующая модель. Для этого мы используем образец проектирования State.

С помощью образца проектирования State можно решить следующую задачу: как использовать объект, не зная его состояния. В контексте игры *Встреча* нам необходимо написать код, управляющий действиями мыши, который, однако, не должен содержать ссылок на возможные состояния игры и на определенные побочные действия мыши. Это позволит нам добавить к игре новые игровые ситуации, не нарушая имеющийся управляющий код.

Подойдем к ситуации в самом общем смысле: необходимо использовать метод `doRequest()` объекта `target` класса `Target`. При этом поведение метода `doRequest()` зависит от состояния, в котором находится объект `target`. Можно решить эту задачу так: ввести новый класс `TargetState` и дополнить класс `Target` свойством `targetState` типа `TargetState`. Требуется, чтобы свойство `targetState` всегда должным образом отражало текущее состояние объекта `target`, который является объектом соответствующего подкласса класса `TargetState` (рис. 5.23).

Метод `doRequest()` просто вызывает по очереди методы `targetState.handleRequest()`, таким образом, вызов метода `doRequest()` транслируется свойством виртуальной функции в конкретную версию функции `handleRequest()`, соответствующую состоянию объекта `target`. Все это происходит невидимо для клиента, вызывающего метод `doRequest()`. Другими словами, клиенту не требуется знать состояние объекта `target`.

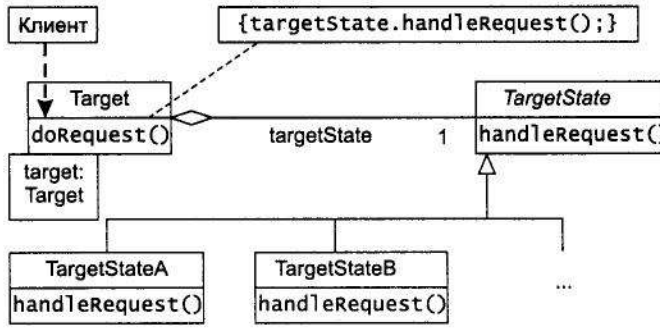


Рис. 5.23. Структура образца проектирования State: поведение метода doRequest() зависит от состояния объекта target

Применение образца проектирования State к проекту *Встреча* для управления состояниями и действиями видеоигры иллюстрирует рис. 5.24. Каркасный класс *РолеваяИгра* имеет свойство под названием *состояние* типа *СостояниеИгры*. Тип *состояния* (от которого зависит, к какому подклассу класса *СостояниеИгры* данное свойство принадлежит) определяет, что происходит, когда вызывается метод `handleEvent()` в объекте типа *РолеваяИгра*. Программный код метода `handleEvent()` в классе *РолеваяИгра* передает управление функции состояния `handleEvent()`. Каждый подкласс класса *СостояниеИгры* выполняет функцию `handleEvent()` по-своему. Например, если игра *Встреча* находится в состоянии *Установка характеристик*, а событие заключается в появлении внешнего персонажа, то окно, в котором пользователь задает значения характеристик персонажа, исчезает, потому что именно так запрограммирован метод `handleEvent()` в классе *Настройка*. Кроме того, следствием определенного сочетания события и состояния является то, что *Встреча*, в соответствии с диаграммой переходов состояний, переходит в состояние *Контакт*. Такой переход осуществляется с помощью кода, например, такого вида:

```
ИграВстреча.setState(new Контакт());
```

Когда в игре произойдет следующее событие, будет выполнена функция `handleEvent()` класса *Контакт*, поскольку сейчас игра находится в состоянии *Контакт*.

Архитектура образца проектирования State особенно выгодна, когда велика вероятность того, что в будущем потребуется добавить в систему новые состояния. Недостатком этого образца проектирования является то, что он не способен решить вопрос: какое состояние необходимо установить после того, как будет выполнена функция `handleEvent()`. Альтернатива образца проектирования State заключается в использовании простой таблицы «состояние—действие», записи которой указывают, какие действия следует совершить, если приложение находится в данном состоянии и происходит данное событие.

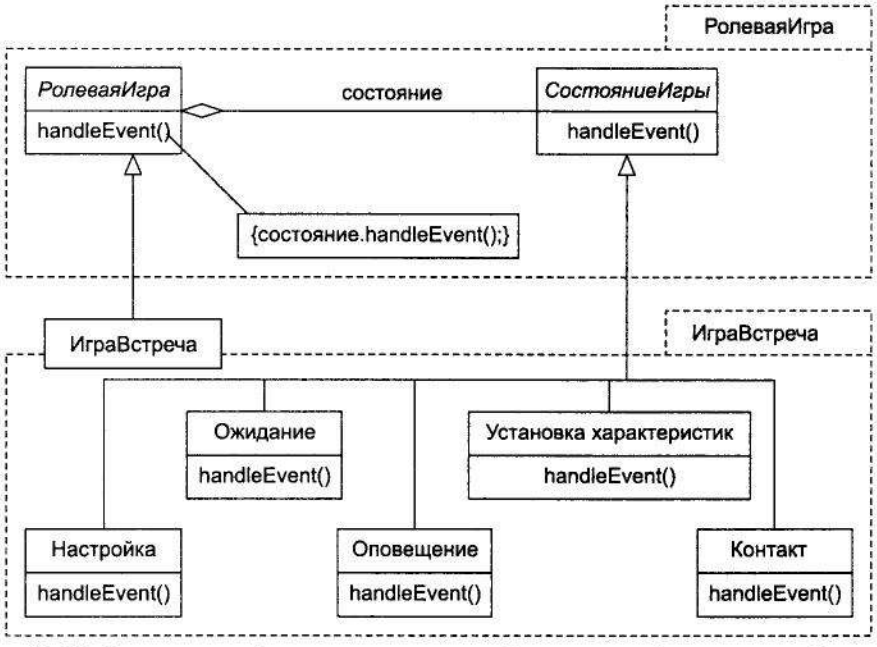


Рис. 5.24. Применение образца проектирования State к ролевой игре и к игре Встреча

Пример работы с образцом проектирования State приводится в разделе 5.5.2, где также перечислены все «за и против» его применения. Независимо от того, применяется или нет образец проектирования State, архитектуры, ориентированные на состояние, с успехом используются для многих приложений (рекомендуем уделить особое внимание работам [Sh2]). Приложениям, расчитанным на работу в реальном времени, таким как приложение маршрутизации для мобильного телефона, использование архитектуры состояний дает особые преимущества.

5.3.3. Виртуальные машины

Архитектура *виртуальных машин* рассматривает приложение как программу, написанную на специальном языке. Из-за того, что должен быть реализован интерпретатор этого языка, эта архитектура окупается, только если будут написаны несколько программ, генерирующих приложения.

В качестве примера архитектуры виртуальных машин рассмотрим приложение, обрабатывающее заказы определенных компьютерных систем одной сети. Обработка заказов выражается на специальном языке со следующей грамматикой:

```
Program ::= 'assemble' Program | 'price' Program | System
System ::= Computer | '{' System 'and' System '}'
Computer ::= '{' CPU '&' RAM '}'
CPU ::= '260MHz' | '300MHz' | '400MHz'
RAM ::= '32MB' | '64MB' | '12MB'
```

Например, заказ на рис. 5.25 состоит из системы 260 МГц с оперативной памятью 64 Мбайт, которая соединена с системой, содержащей два компьютера: 400 МГц, 128 Мбайт RAM и 260 МГц, 32 Мбайт RAM.

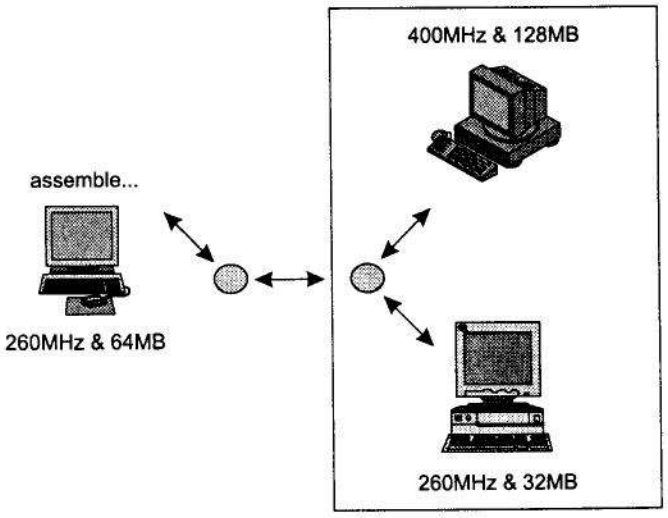


Рис. 5.25. Пример «программы» для виртуальной машины

Этот заказ выражается с помощью описанной грамматики следующим образом:

```
Assemble
{                {
                 { 260MHz & 64MB }
                 }
                 and
                 {        { 400MHz & 128MB }
                          and
                          { 260MHz & 32MB }
                 }
}
```

Это выражение корректно с точки зрения описанной грамматики, как показано ниже.

```
Program
assemble Program
assemble System
assemble { System & System }
assemble { Computer & { System & System } }
assemble { { CPU & RAM } & { Computer & Computer} }
assemble { { 260MHz & 64MB } } & { { CPU & RAM } & { CPU & RAM } }
assemble { { 260MHz & 64MB } } & { { 400MHz & 128MB } & { 260MHz & 32MB } }
```

Вывод, выдаваемый виртуальной машиной, будет иметь вид:

```
1 Construct system 1 as follows:
2           Computer with 260MHz CPU and 64MB RAM
3 Construct system 2 as follows:
    Construct system 3 as follows:
            Computer with 400MHz CPU and 128MB RAM
    Construct system 4 as follows:
            Computer with 260MHz CPU and 32MB RAM
Connect System 3 and System 4
Connect System 1 and System 2
```

Мы могли бы сделать вывод более удобным для использования, но в таком виде показанный нами вывод упрощает пример.

Преимущество использования архитектуры виртуальных машин в том, что вы можете свободно генерировать приложения, выражая их специальным языком. Например, следующий код мог бы быть еще одной программой, выдающей цену на основе обработанных инструкций:

```
price assemble { ... }
```

Очевидно, что можно сделать этот язык более мощным, а также заменить текстовое представление на графическое. Но и в этом случае применялась бы та же архитектура виртуальных машин.

Реализация полной виртуальной машины требует построения интерпретатора. В общем случае интерпретация вызывается операцией — назовем ее `interpret()` — над программой на нашем языке. Интерпретация простейшего одиночного элемента (например, `CPU`) в общем проста (например, это может означать требование вывести `take CPU in the example`). Проблема состоит в запуске `interpret()` при применении к более сложной программе.

Один из путей решения — использовать образец проектирования *Interpreter*. Гамма [33] указывает, что целесообразно использовать этот образец проектирования, когда грамматика мала и скорость не является важным фактором. Это справедливо для рассматриваемого нами примера. Образец проектирования Interpreter показан на рис. 5.26.

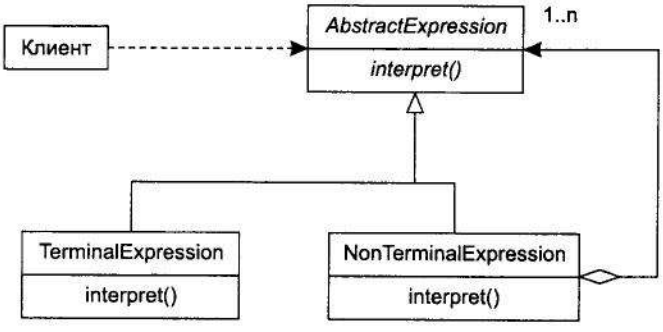


Рис. 5.26. Образец проектирования Interpreter

Объекты AbstractExpression являются либо объектами TerminalExpression, для которых функция интерпретации проста, либо объектами NonTerminalExpression. Последние агрегируют один или более объектов AbstractExpression. Функция interpret(), вызываемая для объектов NonTerminalExpression, выполняется, вызывая для каждого агрегированного объекта AbstractExpression свою версию interpret().

В качестве примера рассмотрим архитектуру интерпретатора нашей грамматики, для простоты игнорируя расчет стоимости. Применяя образец проектирования Interpreter, мы получим диаграмму (рис. 5.27). Ограничимся сборкой только двух компонентов, то есть пусть класс System агрегирует только два объекта Component. Этот метод легко может быть расширен и для большего количества компонентов.

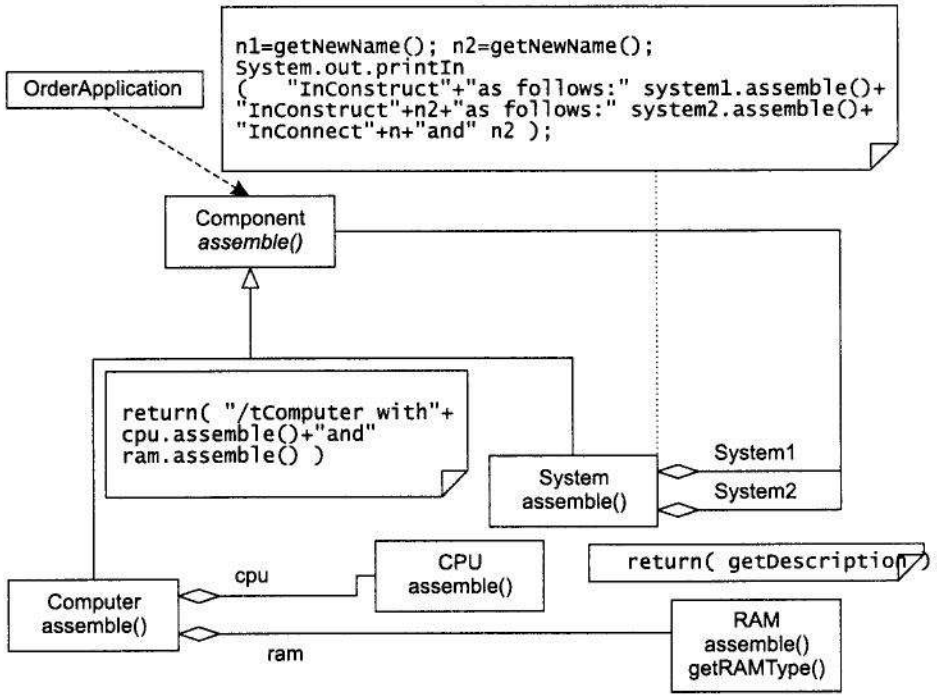


Рис. 5.27. Приложение, построенное на основе образца проектирования Interpreter

Заказ клиента из предыдущего фрагмента программы будет преобразован к следующему виду:

```
assemble( order )
```

где order — это объект System:

```
System order = new System (
// System1 задает параметр первого конструктора:
new Computer( 260, 64 ),
```

```
// System2 задает параметр второго конструктора;
new System( new Computer ( 400, 128 ), new Computer (260, 32) )
);
```

Команда `assemble( order )` объекта `Component` возвращается к методу `assemble()` объекта `System`. Он создает строку 1 в выводе и вызывает `assemble()` для каждого из двух своих параметров (см. рис. 5.27). Для первого параметра, объекта `Computer`, `assemble()` выдает строку 2. Затем появляется и строка 3. При вызове `assemble()` с третьим параметром выполняются действия над объектом `System`, который, в свою очередь, передает управление двум объектам `Computer`, и т. д.

Архитектура виртуальных машин очень полезна, если приложение состоит из процессов со сложными элементами (например, заказы), которые можно выразить с помощью грамматики. В качестве дополнительного примера можно привести приложение, выполняющее простое программирование на уровне пользователя на специальном языке. Обычный пользователь может написать, например, такой сценарий:

```
Проверка баланса / добавить сумму на счет + вычесть дефицит из остатка;
Сохранить отчет / c:Reports + стандартные заголовки + заменить «Ed» на «Al»
Распечатать отчет / стандартные заголовки
Отправить отчет по адресу Jayne@xyz.net
```

Архитектура виртуальных машин анализирует и интерпретирует такие сценарии.

5.3.4. Репозиторные архитектуры

Архитектура, построенная главным образом вокруг данных, называется *репозиторной* архитектурой. Большинство таких систем предназначены для обработки транзакций по отношению к базам данных. Предположим, например, что электрическая компания имеет базу данных. В ней хранятся данные о клиентах: месячное потребление энергии, баланс, история платежей, ремонты и т. д. Типичными операциями для этой базы данных являются добавление нового клиента, операция кредитования, запрос на историю платежей. Типичный проект для репозиторных архитектур такого рода представлен на рис. 5.28.

Другие примеры репозиторных архитектур можно найти в области интерактивных сред разработки (IDE — Interactive Develompent Environment). Интерактивные среды разработки широко применяют такие процессы, как редактирование и компиляция в базу данных исходных кодов и объектных файлов.

Существует масса литературы, посвященной архитектурам баз данных, и в этой книге мы не будем пытаться охватить весь материал по данной теме. Многие приложения, к примеру IDE, не затрагивают базы данных, пока последние не закончены. Проект игры *Встреча* в его простейшей форме не содержит базы данных. Однако если игра вырастет до размеров нескольких десятков персонажей, то может оказаться предпочтительней хранить персонажи в базе данных, а не в специальных файлах. И уж точно нам не обойтись без базы данных, если мы разрешим пользователю запрашивать статистику, например «количество персо-

нажей с силой более 10». Для выражения подобных запросов используется язык SQL (Structured Query Language — язык структурированных запросов) [72].

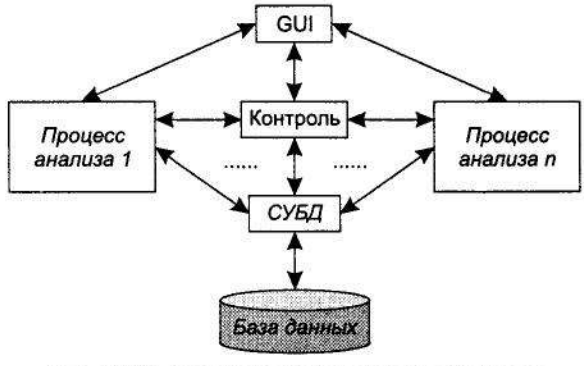


Рис. 5.28. Типичная репозиторная архитектура

Архитектуры *досок объявлений*, разработанные для приложений с искусственным интеллектом, являются *репозиториями* с определенными правилами поведения. Детальное рассмотрение архитектур досок объявлений приведено в [66].

И последней из репозиторных архитектур, которые мы рассмотрим, будет *гипертекстовая* архитектура. Использование этой архитектуры широко распространено в Web. Другим примером могут служить приложения, управляющие документацией по разработке программного обеспечения (см. рис. 1.19).

Слово «репозиторий» зачастую используется для указания того, что приложение предоставляет унифицированный интерфейс доступа к коллекции баз данных. Репозитории не изменяют структуры баз данных, а только предоставляют унифицированный доступ к ним. Этот особый случай репозиторных архитектур определен Гарланом и Шоу в [34].

Репозиторные архитектуры используются в огромном количестве приложений, поскольку базы данных, которые нужно поддерживать, присутствуют во многих архитектурах. Когда обработка незначительно отличается от форматирования данных из базы данных, репозиторные архитектуры являются самыми подходящими. С другой стороны, наличие большой базы данных иногда свидетельствует о том, что значительный объем обработки данных может влиять на архитектуру. В приложениях, которые изначально не следовало проектировать с помощью репозиторной модели, быстро распространяется программирование *ad hoc*[1] (например, с использованием хранимых процедур).

5.3.4.1. «Визит» к членам репозитория образца проектирования Iterator

Репозиторные архитектуры находят применение в приложениях, где необходима агрегация (коллекция) объектов. Агрегация представлена в таком виде, чтобы ее элементы можно было «посетить». В случае, когда агрегированные объекты

[1] Специально для данного случая (*лат.*).

могут «посещаться» несколькими способами, целесообразно использовать образец проектирования *Iterator*. Примером может служить система управления персоналом, требующая выдавать список персонала различными способами. Предположим, что объекты служащих хранятся в виде дерева, согласно организационной диаграмме. Клиентское программное обеспечение для множества служащих может быть реализовано так, что посещение структуры данных будет осуществляться способом, определенным параметром `Iterator`. Этот параметр выполняет операции над каждым индивидуальным объектом, такие как распечатка имен, запись всего досье служащего в файл. Посещение может происходить различными способами: в алфавитном порядке, по должности и т. д.

Предположим, что `agg` — это агрегация объектов класса `C`. Объекты `Iterator`, посещающие `agg`, должны иметь следующие четыре функции.

- `Iterator` «указывает» на первый элемент:
  ```
  void setToFirst();
  ```
- `true`, если `Iterator` «указывает» на последний элемент:
  ```
  boolean isDone();
  ```
- Превращает `Iterator` в указатель на следующий элемент:
  ```
  void increment();
  ```
- Возвращает элемент, на который указывает `Iterator`:
  ```
  C getcurrentElement();
  ```

После того как объекты `Iterator` определили эти четыре функции, мы можем создавать использующий их код. Таким образом, мы выполняем операции агрегации во время исполнения в том порядке, в каком они определены в объекте `Iterator`:

```
/*
Выполняем желаемую операцию над элементами агрегации в соответствии с номером
итерации (i):
*/
for(i.setToFirst(); i.isDone(); i.increment())
операция(i.getcurrentElement());
```

Мы могли бы не использовать Iterator, если бы посещения происходили только одним способом.

5.3.5. Уровневые архитектуры

Уровень архитектуры — это логически связанная коллекция артефактов программного обеспечения, обычно — пакеты классов. В общем виде уровень использует не более одного уровня и вместе с тем используется не более чем одним уровнем. Построение приложений последовательно уровень за уровнем сильно упрощает процесс. Некоторые уровни, например каркасы, могут использоваться в нескольких приложениях.

Мы уже видели, как уровневый подход применяется к приложению *Встреча*. В этом случае классы в пакетах наследуются от классов в каркасных пакетах. Еще один пример — использование ускорителя трехмерной графики в качестве уровня, доступного с уровня ролевой игры *Встреча* — показан на рис. 5.29.

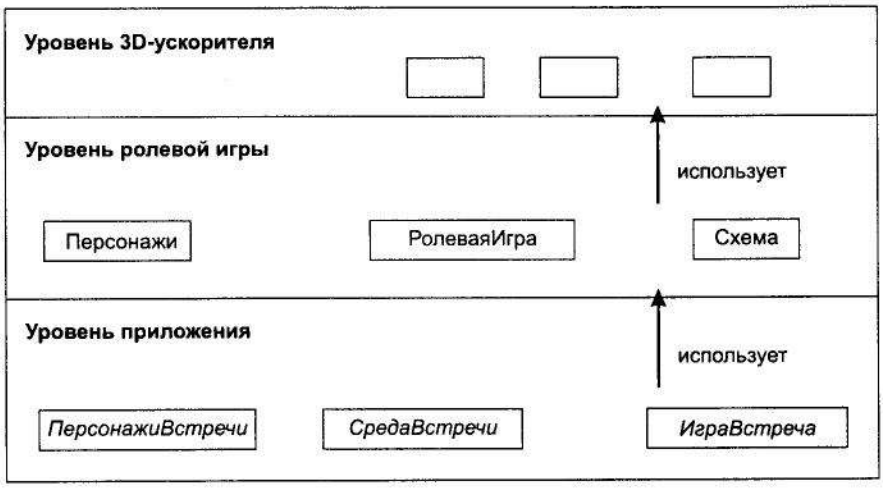


Рис. 5.29. Уровневые архитектуры

Пример многоуровневой архитектуры для банковского приложения печати Ajax представлен на рис. 5.30. В последнем случае имеется четыре уровня архитектуры. Зависимости между уровнями архитектуры показаны на рис. 5.31 в порядке, обратном используемому на рис. 5.30. Уровень приложения *Ajax Bank Printing* должен отвечать за печать и форматирование. Он построен на основе уровней *Accounts* и *Ajax Bank Common Library*. Последние построены на основе уровня, поддержку которого осуществляет продавец (на рис. 5.30 не показан). Этот уровень содержит общие утилиты, такие как средства сортировки и поиска. Обычно уровень реализуется в виде пакета классов. Например, *Ajax Bank Common Library* включает в себя классы, используемые в Ajax-приложениях. В качестве отношений может использоваться наследование, агрегация или объектная ссылка. Например, между уровнями допустима только агрегация.

Клиент-серверная архитектура является общей уровневой формой. В этой форме уровень клиента связывается с серверным уровнем для получения требуемого сервиса. Клиент обычно находится резидентно на компьютере пользователя, а сервер — на более крупном централизованном компьютере. Сервер нередко ссылается на базу данных.

Классическая клиент-серверная архитектура обычно страдает от реализации. Дело в том, что зачастую клиенты и серверы закодированы на уровне технических средств в высокой степени зависимости друг от друга. Эта проблема может быть решена с помощью трехуровневой архитектуры, в которой средний уровень предназначен для разделения клиента и сервера. Средний уровень можно использовать для увеличения гибкости архитектуры несколькими способами. Например, если несколько серверов одновременно могут обслужить запрос клиента,

то средний уровень может динамически определить подходящий сервер. Средний уровень обычно реализуется в виде *связующего программного обеспечения* (middleware). Стандартом такого программного обеспечения является архитектура CORBA, разработанная консорциумом OMG.

Рис. 5.30. Пример уровневой архитектуры с использованием агрегации

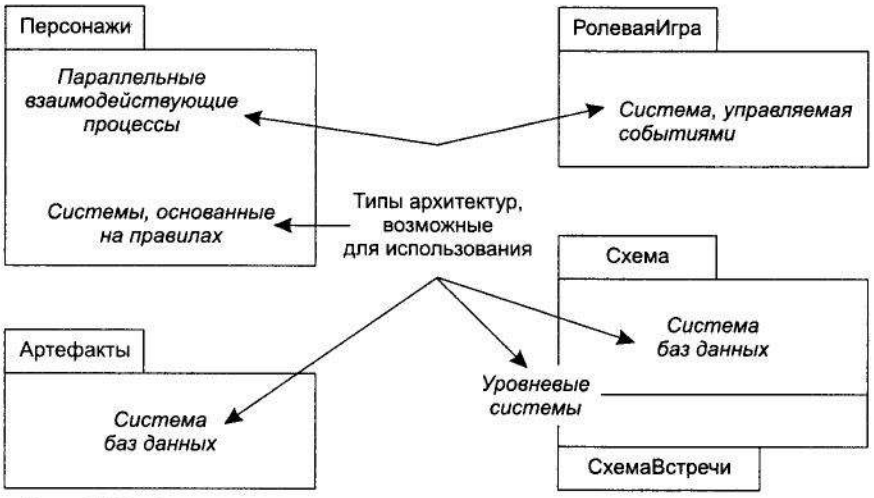


Рис. 5.31. Возможное использование уровневой архитектуры в игре Встреча

Уровневые архитектуры имеют множество преимуществ при повторном использовании. Библиотека классов Java является очень эффективной уровневой системой (пакет `applet` — уровень — основан на пакете `awt`, который в свою очередь основан на пакете `lang`, и т. д.).

5.3.6. Приложения со смешанной архитектурой

Приложения обычно используют несколько архитектур. Например, каркас для ролевых видеоигр может использовать несколько типов архитектур, приведенных Гарланом и Шоу (см. рис. 5.31). Это будет иметь смысл, например, если организовать пакет *Артефакты* как базу данных. Персонажи игры могут рассматриваться как параллельные взаимодействующие процессы; систему управления игрой мы можем организовать как систему обработки событий.

5.3.7. Подведение итогов: процедура выбора архитектуры

Подведем итоги нашему обсуждению выбора и разработки архитектуры.

ОДИН ИЗ СПОСОБОВ ВЫБОРА АРХИТЕКТУРЫ (1)

1. Разбейте систему на замкнутые модули.
2. Сравните со стандартными архитектурами (из классификации Гарлана и Шоу). Улучшите декомпозицию.
   - Есть поток данных в пакетах между обрабатывающими станциями?

     Архитектура последовательных пакетов.
   - Обрабатывающие станции ожидают получения входных данных, чтобы начать свою работу?

     Архитектура каналов и фильтров.
   - Процессы выполняются параллельно?

     Архитектура параллельных взаимодействующих процессов.
   - Процесс обеспечивает обслуживание пользовательских процессов?

     Клиент-серверная архитектура.
   - Процесс реагирует только на происходящие события?

     Системы, управляемые событиями.
   - Приложение состоит из процессов, которые выполняются по сценарию?

     Образец проектирования Interpreter.
   - Приложение строится для хранилища данных?

     Репозиторные архитектуры.
   - Существует упорядочение по уровням?

     Уровневые архитектуры.

ОДИН ИЗ СПОСОБОВ ВЫБОРА АРХИТЕКТУРЫ (2)

3. Сделайте выбор среди представленных альтернативных архитектур.
4. Добавьте к классам, полученным на основе анализа требований, классы, обеспечивающие согласование с выбранной архитектурой.

   Например, в системе, управляемой событиями, это могут быть классы, контролирующие переходы между состояними.
5. Примените существующий каркас и (или) образец проектирования, если найдете полезный.
6. Распределите классы по пакетам.

   В идеале должно быть 4–8 пакетов (для больших приложений используйте вложенные пакеты).

   Каждый пакет должен иметь смысл в контексте приложения (например, пакет *Видеофильмы* допустим, а пакет *БольшиеКлассы* — нет).
7. Удостоверьтесь, что связность между частями высока. Низкое сцепление будет подтверждением правильного выбора.
8. Рассмотрите возможность добавления facade-класса (объекта) для управления интерфейсами пакетов.

5.4. Архитектура: нотация, стандарты и инструментальные средства

5.4.1. Нотация

UML — это широко распространенная графическая нотация для изображения объектно-ориентированных проектов. В этой книге освещаются некоторые аспекты UML.

Мы рассмотрели нотацию переходов состояний в разделе 3.3.4 и диаграммы потоков данных в разделе 5.3.1. Они могут применяться независимо от объектной ориентированности. Это верно и для диаграмм «объект—отношение», показывающих отношения между данными в хранилищах данных.

5.4.2. Инструментальные средства

Для облегчения процесса разработки программного обеспечения используется множество автоматизированных инструментальных средств. Некоторые из них представляют собой коллекцию классов с различными взаимосвязями. Примерами таких коллекций могут служить *Rational Rose* от Rational Corporation и *Together* от Object International. Эти инструменты облегчают построение объектных моделей, а также их соединение с соответствующим исходным кодом и диаграммами последовательности.

Для выбора инструментальных средств моделирования составляется список требований к ним. Этот процесс аналогичен процессу анализа требований для

разработки программного приложения. Приведем список некоторых требований к инструментам моделирования.

- [необходимо] Облегчение изображения объектных моделей и диаграмм последовательности.
  - Быстрое создание классов.
  - Легкое редактирование классов.
  - Изменение масштаба изображения внутри частей модели.
- [желательно] Возможность быстрого перехода от объектной модели к исходному коду.
- [необходимо] Должен стоить не более $X для одного пользователя.
- [не обязательно] Возможность обратного проектирования (то есть создания объектной модели из исходного кода).

5.4.2.1. Высокоуровневые и низкоуровневые инструментальные средства

Пакеты инструментальных средств зачастую пытаются охватить и архитектуру, и детальное проектирование, и реализацию. Различные продавцы разрабатывают системы с возможностью использования гиперссылок между исходным кодом и документацией. Инструментальные средства, ориентированные на реализацию, подобные Javadoc, могут быть хорошим дополнением для процесса разработки. Клиент-серверные инструменты, такие как Powerbuilder, вполне пригодны для определения архитектуры, хотя они определяют и реализации. Javadoc очень полезен при навигации по пакетам, поскольку он предоставляет алфавитный список всех классов, а также их иерархию.

Интерактивные среды разработки (IDE) укомплектованы компиляторами и используются как инструменты частичного моделирования. Объектно-ориентированные IDE в основном показывают наследование в иерархической форме. Этот факт привлекает разработчиков из-за близости этих инструментов к процессам компиляции и отладки. Однако IDE обычно имеют недостаточно широкий спектр возможностей, чтобы облегчить построение архитектуры и проектировочную работу.

Инструментальные средства сборки компонентов позволяют создавать приложения с помощью перетаскивания значков, представляющих элементы процессов. Среды JavaBeans представляют собой пример таких инструментальных средств. В таких средах объекты Java, классы которых соответствуют стандарту Java Beans, можно взять из библиотек или создать самостоятельно и связать через события. Стандарт Java Beans был создан для облегчения таких простых сборок с помощью графических инструментальных средств.

Основное неудобство при использовании инструментов моделирования связано с зависимостью проекта от третьей стороны — продавца. Вдобавок ко всему, кроме сложности самого приложения и проекта разработчик должен беспокоиться о жизнеспособности продавца. Если продавец обанкротится или обновление для инструментальных средств станет слишком дорогим — как это отразится на проекте?

Но несмотря на все это популярность инструментов моделирования растет во всем мире. Продолжительность их использования ограничивается продуктивностью и экономическими факторами.

5.4.3. Стандарт IEEE/ANSI для описания проекта

Стандарт IEEE 1016-1987 (вновь утвержденный в 1993 году) для проектной документации программного обеспечения (SDD — Software Design Document) содержит руководство по составлению и ведению документации по разработке. Оглавление этого документа представлено далее. Руководство, включенное в стандарт IEEE 1016.1-1993, объясняет, каким образом SDD может быть составлена для различных стилей архитектур. Большинство из них мы уже рассмотрели. Упражнения в конце главы используют стандарт IEEE с некоторыми модификациями для подчеркивания выразительности объектно-ориентированной парадигмы. Разделы стандарта 1–5 могут быть отнесены к архитектуре программного обеспечения, а раздел 6 — к детальному проектированию.

1. **Введение**
   - 1.1. Цель
   - 1.2. Описание проекта
   - 1.3. Определения, сокращения и термины
2. **Ссылки**
3. **Описание декомпозиции**
   - 3.1. Модульная декомпозиция
     - 3.1.1. Описание модуля 1
     - 3.1.2. Описание модуля 2
   - 3.2. Декомпозиция на параллельные процессы
     - 3.2.1. Описание процесса 1
     - 3.2.2. Описание процесса 2
   - 3.3. Декомпозиция данных
     - 3.3.1. Описание блока данных 1
     - 3.3.2. Описание блока данных 2
4. **Описание зависимостей**
   - 4.1. Межмодульные зависимости
   - 4.2. Межпроцессные зависимости
   - 4.3. Зависимости внутри данных
5. **Описание интерфейса**
   - 5.1. Модульный интерфейс
     - 5.1.1. Описание модуля 1
     - 5.1.2. Описание модуля 2

5.2. Интерфейс процессов
5.2.1. Описание процесса 1
5.2.2. Описание процесса 2

6. Детальное проектирование

6.1. Детальное проектирование модулей
6.1.1. Модуль 1: детали
6.1.2. Модуль 2: детали
6.2. Детальное проектирование данных
6.2.1. Блок данных 1: детали
6.2.2. Блок данных 2: детали

5.5. Контроль качества при выборе архитектуры

Персонал, осуществляющий контроль качества, должен принимать участие в оценках архитектуры. Кроме того, он разрабатывает планы тестирования для всех компонентов архитектуры, начиная с того момента, как эти компоненты определены.

5.5.1. Качество и выбор архитектуры

5.5.1.1. Метрики для выбора архитектуры

Большинство приложений могут быть реализованы с помощью различных архитектур. Некоторые варианты могут быть намного лучше других. Такие важные решения, как выбор архитектуры, не принимаются без первичной разработки и сравнения альтернатив. Предложенные архитектуры тщательно анализируются, поскольку устранение дефектов на ранней стадии стоит намного меньше, чем их исправление во время реализации проекта.

В этом разделе мы предложим метрики для выбора архитектуры, а в следующем рассмотрим примеры выбора наиболее подходящей архитектуры.

Один из способов выбора — присвоить веса требуемым характеристикам и назначить нечеткий коэффициент качества для каждого кандидата (табл. 5.3). Таблица 5.3 может быть использована для сравнения вариантов во многих областях при условии, что критерии могут быть выбраны и взвешены. Следующие метрики из [59] применяются к проектированию программного обеспечения почти повсеместно.

«13. Количество входов и выходов для модуля (пакета). Эта величина может быть вычислена путем учета количества доступных извне модуля открытых методов. Количество выходных точек вычисляется путем подсчета открытых функций, которые возвращают значения вызвавшему их объекту или произво-

дят изменения в среде вне этого модуля (пакета). Основной целью является минимизация этой величины для уменьшения количества взаимодействий».

«15. Графико-теоретическая сложность архитектуры. Простейшей (статической) версией этой метрики является выражение:

Количество модулей в архитектуре – Количество модулей, имеющих хотя бы один вызов функции между ними + 1».

Таблица 5.3. Нечеткий метод сравнения архитектур

Характеристика	Архитектура 1	Архитектура 2	Архитектура 3
Качество (Вес: 1–10)	9 = Высокое; 5 = Среднее; 2 = Низкое		
Расширение (e)	ea1	ea2	ea3
Изменение (c)	ca1	ca2	ca3
Простота (s)	sa1	sa2	sa3
Эффективность: скорость (esp)	espa1	espa2	espa3
Эффективность: хранение (est)	esta1	esta2	esta3
Итого:	e*ea1+c*ca1+s*sa1+ +esp*espa1+est*esta1	e*ea2+c*ca2+s*sa2+ +esp*espa2+est*esta2	e*ea3+c*ca3+s*sa3+ +esp*espa3+est*esta3

Применим метрику 15 к архитектуре системы, имитирующей работу банка (рис. 5.32). Архитектура делит имитацию на следующие пакеты:

- `SimConfiguration` — определяет способ распределения средств между станциями внутри банка;
- `SimItems` — определяет объекты, которые перемещаются внутри банка;
- `SimEvents` — обрабатывает настоящие и будущие события, имеющие место в банке (то есть прибытие клиента к окну обслуживания);
- `Simulation` — механизм, управляющий имитацией (выбирает следующее событие для исполнения, исполняет его, обрабатывает последовательности, включая генерацию результирующих событий, и составляет очередь из них в объекте `SheduledEvents`);
- `Random` — пакет, осуществляющий генерацию чисел на основе различных распределений (например, генерирует длительность обслуживания следующей транзакции).

Эта архитектура разработана с помощью образца проектирования Facade. Для расчета метрики будем учитывать ссылки только между пакетами. В архитектуре имеется пять узлов (пакетов), а также пять пар модулей, между которыми имеются вызовы функций (в любую сторону). Таким образом, сложность статической архитектуры 5 – 5 + 1 = 1. Это число также характеризует количество циклов внутри архитектуры (его можно вычислить непосредственно как 1 – `SimDriver/Events/Configuration`). В результате мы выясняем, что данная архитектура не является сложной, что, в общем-то, хорошо.

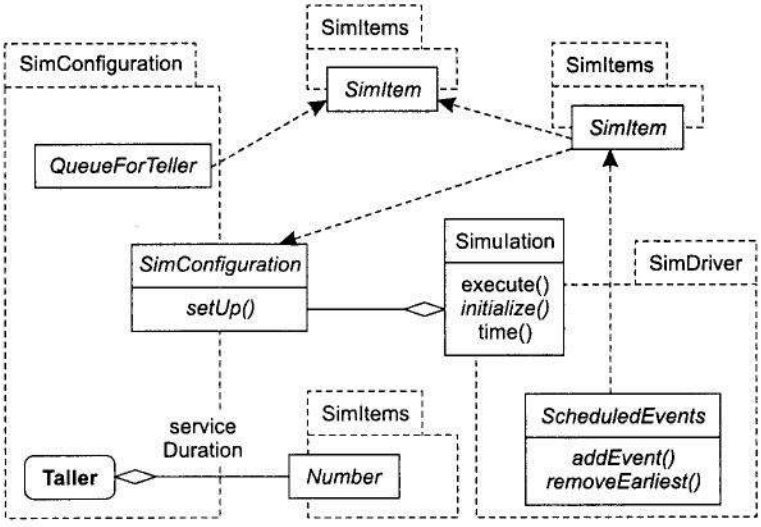


Рис. 5.32. Архитектура системы, имитирующей работу банка

«**25. Сложность потока данных или информации.** Этот показатель измеряет поток информации в крупномасштабных объектах, сложность потоков в процедурах и модулях, а также сложность соединений между модулями. Детальное определение находится в метрике 4.25 [59]».

Метрики наподобие этой предусматривают квантификацию. Но каким же образом использовать получившиеся числа? Один ответ мы можем найти в истории. Например, мы можем легко сказать, что пакет *ИграВстреча* на данный момент имеет четыре открытых функции (см. упражнения в конце главы). Возможно, мы можем предсказать, что количество этих функций в дальнейшем возрастет до величины от 10 до 15. Эти числа сравниваются с соответствующими средними значениями предыдущих проектов. Если среднее число 10 и мы удовлетворены декомпозицией предыдущих проектов, то полученная нами величина не вызывает никаких опасений. Если же среднее число 8, а мы получили 15, то следует внимательнее рассмотреть выбранную архитектуру.

5.5.2. Выбор из альтернативных архитектур

Мы не будем подробно разрабатывать одну архитектуру при выборе из подходящих вариантов архитектур. В качестве примера рассмотрим архитектуру приложения *Встреча*.

- Вариант 1 для приложения *Встреча* — образец проектирования State. Он является возможной архитектурой для нашего приложения (мы обсуждали это в разделе 5.3.2.3). Мы будем сравнивать эту архитектуру с другим кандидатом. В качестве альтернативы возьмем архитектуру, рассмотренную в разделе 5.3.2.3 как вариант 1.

- Вариант 2 для приложения *Встреча — специальная архитектура под управлением графического интерфейса пользователя* (GUI). Вторую архитектуру можно совместить с идеей переходов состояний, а затем писать код обработки событий отдельно для каждого объекта GUI, который чувствителен к событиям мыши и клавиатуры. Такая архитектура показана на рис. 5.33. Для улучшения понимания на рисунке отображены выбранные методы. Выходные гиперссылки для этой архитектуры — это GUI представления объектов *СоединениеЗоны*, а каждому соединительному звену сопоставлен код обработки событий. Например, при выборе мышью выхода в зону *подвал* на экране должно появиться изображение этой зоны. Класс гиперссылок на зону *подвал* должен быть связан с соответствующим обработчиком события. Результат проектирования окажется более управляемым со стороны графического интерфейса пользователя, более специфичным и более зависящим от языка. Существует некоторое разделение между кодом для этого проекта и для компонентов, зависящих от состояния. С другой стороны, классовая модель содержит меньше классов.

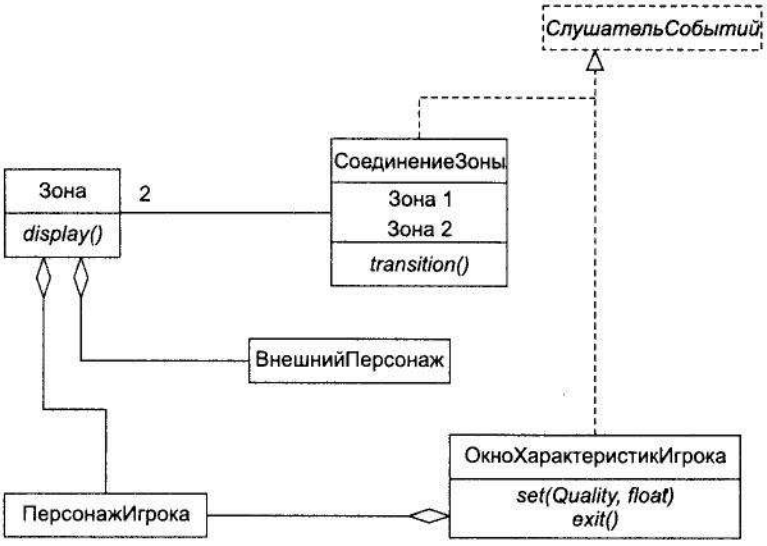


Рис. 5.33. Специальная архитектура под управлением графического интерфейса пользователя для игры Встреча

- Вариант 3 для приложения *Встреча — таблица переходов состояний*. Третий вариант архитектуры основан на идее переходов состояний, но переход состояний будет реализован в виде таблицы. Такой вариант рассматривали Шлаер и Меллор в [99]. Примером управляющей таблицы является табл. 5.4. Эта архитектура использует концепцию переходов состояний, но реализует ее без образца проектирования State.

Таблица 5.4. Обработка событий перехода между состояниями, управляемых таблицей

Текущее состояние	Событие				
	Щелчок мышью для выхода	**Запрос на изменение характеристики**	**Закрытие окна установки характеристик**	**Входит внешний персонаж**	**Уходит внешний персонаж**
Ожидание	Отображение персонажа игрока в соседней зоне	Показать окно установки характеристик		Показать оба персонажа и перейти в состояние Контакт	
Контакт					Вычислить результаты контакта и перейти в состояние Ожидание
Установка характеристик			Убрать установки окно характеристик и перейти в состояние Ожидание	Убрать окно установки характеристик и перейти в состояние Контакт	Перейти в состояние Ожидание

Теперь мы можем вычислить значения метрик, рассмотренных ранее для этой архитектуры. Существует список аргументов «за» и «против» для сравнения этих двух подходов. Более полное сравнение трех архитектур приведено в табл 5.5.

«За» использование образца проектирования State:

- дает возможность легко добавлять или изменять состояния в соответствии с изменениями в проекте игры;
- поясняет, какие действия должны быть выполнены в различных обстоятельствах;
- классифицирует все события мыши, имеющие какое-либо воздействие на игру *Встреча*.

«Против» использования образца проектирования State:

- классовая модель более запутанна и изначально сложна для понимания.
- дублирование данных: состояние игры *Встреча* может быть получено из переменных, не относящихся к объекту State, что увеличивает вероятность ошибки программиста в случае противоречия.

«За» использование таблицы для описания переходов состояний:

- таблица легко понимается и редактируется;
- архитектура не обязательно должна быть реализована на объектно-ориентированном языке;
- возможно документирование этого подхода с использованием метода Шлаера–Меллора [99].

«Против» использования таблицы для описания переходов состояний:

- требует глобальную структуру данных (таблица);
- пополнение таблицы новыми состояниями может нарушить существующий код и проект.

Сравнение трех архитектур при помощи метода сравнения из табл. 5.3 иллюстрирует табл. 5.5. Для свойств расширяемости и изменяемости были выбраны наибольшие веса, и поэтому архитектура с использованием образца проектирования вышла на первое место. Несмотря на использованные метрики, опытные команды стараются рассмотреть подробнее каждую архитектуру независимо.

Таблица 5.5. Пример использования нечеткого метода сравнения архитектур

Характеристика		**Варианты архитектуры**		
		1. Образец проектирования State	**2. Специальный управляемый графическим интерфейсом пользователя**	**3. Таблица переходов состояний**
Качество	Вес 1–10	Высокий = 9; Средний = 5; Низкий = 2		
Расширение	9	Высокий	Низкий	Средний
Изменение	7	Высокий	Низкий	Средний
Простота	5	Низкий	Высокий	Средний
Эффективность: скорость	5	Средний	Высокий	Средний
Эффективность: хранение	2	Низкий	Низкий	Средний
Итого (чем больше, тем лучше):		183	126	140

5.5.3. Проверка архитектуры с помощью вариантов использования

Варианты использования получают из требований заказчика. Поэтому варианты использования не могут учитывать архитектуру приложения по простой причине — она еще не определена. После выбора архитектуры полезно вернуться к рассмотрению вариантов использования и проверить, адекватно ли их поддерживает архитектура. Например, вариант использования «Войти в контакт с внешним персонажем» (см. рис. 3.6) должен исполняться в архитектуре, которую мы разработали в этой главе (см. рис. 5.10). Поскольку на протяжении всего проекта мы сохраняем классы предметной области, классы, на которые мы ссылались в вариантах использования, должны быть представлены среди используемых. Обычно диаграммы последовательности для вариантов использования на этом этапе включают дополнительные классы архитектуры.

5.5.4. Инспектирование выбора архитектуры

Архитектуры проверяются на соответствие требованиям. Напомним, что выигрыш во времени и средствах максимален, если дефект обнаружен и устранен на начальной стадии проекта, например на уровне выбора архитектуры. Метрики, рассмотренные ранее, составляют один из базисов инспектирования архитектуры.

Инспектирование пакетов каркаса архитектуры для игры *Встреча* может привести к выводу, что еще нет никаких требований для появления пакета *Артефакты* и что этот пакет и есть собственно дефект.

В качестве дополнительного примера возьмем диаграмму переходов состояний, изображенную на рис. 5.34. Внимательное рассмотрение диаграммы переходов состояний может выявить дефекты, показанные на рисунке. Эти дефекты можно устранить путем уточнения имен и (или) определений событий, на которые выполняются ссылки.

Рис. 5.34. Дефекты в диаграмме переходов состояний для игры Встреча

5.5.5. Влияние выбора архитектуры на SPMP

Теперь, когда мы определились с архитектурой, план-график можно сделать более определенным и детализированным. Также может быть определен порядок построения различных частей проекта. Например, имеет смысл разработать каркасный пакет *Персонажи*, прежде чем создавать пакет приложения *ПерсонажиВстречи*. Поскольку эти пакеты не будут закончены на первой итерации спирали, мы обозначим соответствующие задачи как *Персонажи I* и *ПерсонажиВстречи I*. Стрелками

обозначим зависимость одного пакета от другого. Например, *ПерсонажиВстречи I* невозможно завершить, пока не закончена задача *Персонажи I* (рис. 5.35). План-график показывает, что *Интеграция и Тестирование I* не могут начаться до тех пор, пока не будут завершены все остальные задачи первой итерации.

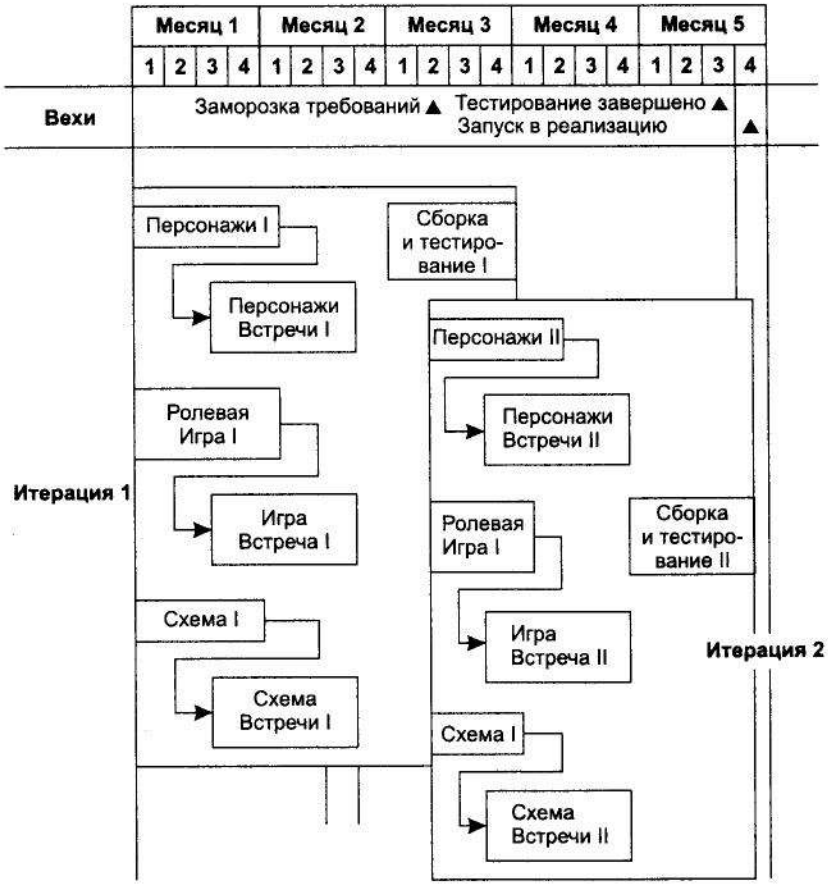


Рис. 5.35. План-график проекта, модифицированный после выбора архитектуры

5.6. Подведение итогов

Термин *архитектура программного обеспечения* имеет отношение ко всему процессу проектирования приложения на самом высоком уровне. Ниже представлена классификация архитектур программного обеспечения по Гарлану и Шоу [34].

- Архитектуры потоков данных.
  - последовательные пакеты;
  - каналы и фильтры.

- Независимые компоненты:
  - параллельные взаимодействующие процессы;
  - клиент-серверные системы;
  - системы, управляемые событиями.
- Виртуальные машины:
  - интерпретаторы;
  - системы, основанные на правилах.
- Репозиторные архитектуры:
  - базы данных;
  - гипертекстовые системы;
  - доски объявлений.
- Уровневые архитектуры.

Каркас — это коллекция классов, которая применяется к семейству приложений. Образцами проектирования называются многократно используемые комбинации классов, решающие задачи, которые возникают достаточно часто. Некоторые из образцов проектирования могут применяться для построения архитектуры. Стандарт IEEE, который применяется к проектированию, называется SDD (Software Design Document). Выбор архитектуры является важнейшим решением. Он осуществляется между несколькими альтернативными архитектурами, что в общем виде можно выразить так.

- Каркас — это базис семейства приложений.
- Образцы проектирования — это повторно используемые комбинации классов, решающие часто возникающие проектные задачи.
- Стандарт IEEE SDD является полезной отправной точкой.
- Создавайте и сравнивайте различные варианты архитектур.

В литературе вы можете встретить множество определений термина *архитектура программного обеспечения*. Некоторые из них приведены в [6].

Руководство по учебному проекту. Архитектура проекта Встреча

В данном разделе мы рассмотрим принципы и методы, используемые при выборе архитектуры, а также способы представления архитектуры, о которых до сих пор рассказывалось в этой главе. Мы также поговорим о том, как архитектура реализуется на практике. В качестве примера мы будем рассматривать учебный проект видеоигры *Встреча*. Мы также рекомендуем студентам обращаться к врезкам «Один из способов...», которые во многом помогают в процессе обучения.

Этап 1. Подготовка

В соответствии с SPMP, Карен Петерс была лидером команды (а ее заместителем — Эд Браун) и вела проверку, контроль и наблюдение за работой по проектированию. Карен с самого начала хотела разработать две полновесных, детально проработанных альтернативных архитектуры для игры *Встреча* и вынести их обе на предварительное обсуждение командой разработчиков. Она решила предотвратить возможные столкновения между разработчиками и бесполезные выяснения того, кто разработал архитектуру лучше. Карен полагает, что в подобных случаях верх берет эго, а не техническая целесообразность. На ее памяти остались довольно неприятные случаи компромиссов, на которые пришлось пойти, чтобы «примирить» соревнующиеся архитектуры. В результате этого проект получался слабым, и работать с ним приходилось ежедневно месяцами, если не годами. С другой стороны, Карен не хотела создавать обособленную, изолированную архитектуру. Она решила, что вместе с Эдом они выберут кандидатуры архитектур, подробно и углубленно их изучат, а затем представят команде разработчиков типы архитектур на выбор.

Этап 2. Выбор архитектуры

Эл Пруит, поразмышляв о проектировании приложения игры, представил на рассмотрение Карен набросок — приблизительную схему архитектуры, основанной на графическом пользовательском интерфейсе (см. рис. 5.33). Автор отмечает, что данная архитектура будет простой и легкой для реализации.

Эд и Карен заново просмотрели труды Гарлана и Шоу по классификации архитектур, с тем чтобы определить, не подходит ли игре *Встреча* какая-нибудь из названных там архитектур.

Сначала они задались вопросом, можно ли игру *Встреча* определить как *поток данных* от одного элемента обработки к другому. Данными в этом случае были бы положения персонажей и (или) значения их качеств. Однако такой подход не соответствовал их пониманию игры.

После этого, по Гарлану и Шоу, они обратились к архитектурам *независимых компонентов*. Первой из архитектур была архитектура *параллельных коммуникационных процессов*. Карен полагает, что данная архитектура подходит игре: каждый персонаж можно рассматривать как отдельный процесс. Каждый из этих процессов может осуществляться в отдельном потоке, параллельном остальным, и эти потоки могли бы вступать между собой во взаимодействие каждый раз, когда персонажи встречаются друг с другом. Эта архитектура стала кандидатурой для игры.

Затем была рассмотрена схема «клиент-сервер». Однако в нашем случае не совсем ясно, кто будет выполнять роль клиента, и кто — роль сервера. Данная архитектура была признана неприемлемой.

Следующим типом архитектуры в списке были *системы, управляемые событиями*. Эта архитектура была признана подходящей, так как игра отвечает либо на события, инициируемые пользователем, такие как щелчок на гиперссылке

зоны с целью войти в эту зону, либо на появление внешнего персонажа в той зоне, в которой находится персонаж игрока.

Далее были рассмотрены *виртуальные машины*. Основным критерием в данном случае стал вопрос: существует ли при выполнении приложения игры реальная необходимость в интерпретации сценария. В данном случае такой необходимости нет.

Затем рассматривалась возможнсть построения игры на основе *репозитория данных* (репозиторная система). Данные могут быть следующие: значения, относящиеся к персонажам, и значения, относящиеся к статусу игры. Было решено, что эта архитектура вполне применима в случае, если в игре много персонажей и артефактов, поскольку в этом случае преобладает манипуляция большими объемами информации. Поскольку игра *Встреча* не строится вокруг данных, предложение по использованию такой архитектуры было отклонено.

И наконец, была рассмотрена *уровневая архитектура*. Здесь вопрос заключался в следующем: можно ли игру *Встреча* рассматривать как набор групп классов. При этом одна группа использовала бы одну или две другие группы. Карен полагает, что можно сформировать по крайней мере два полезных уровня: один — для ролевых игр в самом обобщенном смысле, другой — для самой игры *Встреча*. Данную архитектуру взяли на заметку. На этом перечень типов архитектур у Гарлана и Шоу закончился.

Получился следующий список возможных архитектур:

- Архитектура, основанная на графическом пользовательском интерфейсе;
- Параллельные коммуникационные процессы;
- Системы, управляемые событиями;
- Уровневая архитектура.

Далее Эд и Карен рассмотрели, какая из перечисленных моделей более всего соответствует архитектуре игры в целом, а какие могли бы выполнять вспомогательные функции. Они пришли к заключению, что уровни — это основной архитектурный принцип, поскольку в игре присутствует общий уровень ролевых игр и уровень игры *Встреча*. Было признано, что событийно-управляемые системы могли бы стать вспомогательной архитектурой по отношению к уровневой. Подробное обсуждение архитектуры параллельных коммуникационных процессов было отложено до этапа разработки персонажей игры. Все архитектурные замыслы отражены на рис. 5.9 в концептуальном ключе.

Архитектура событийно-управляемых систем реализуется посредством состояний и переходов между ними. Далее обсуждался вопрос, какие средства выбрать для описания событий и переходов: образец проектирования State или таблицу типа «состояние—действие». Было решено, что в выборе архитектуры смогут помочь метрики.

При помощи электронной почты они попытались согласовать с программистами выбор архитектуры, обсудив удельный вес критериев выбора (расширение, изменение и т. п., см. табл. 5.5). Было решено узнать мнение программистов относительно критериев до проведения самого собрания. При этом не указывалось, какие архитектуры были выбраны в качестве кандидатур для проекта. После этого, до проведения собрания, они отослали Элу Пруиту сравнительную таблицу,

чтобы удостовериться, что при описании архитектуры не были упущены никакие критерии.

Эл отметил, что выбор архитектуры коренным образом зависит от удельного веса критериев, но сам он на стороне решения и взвешивания, которые будут предложены группой программистов. Карен и Эд составили электронную таблицу (см. табл. 5.4), сделав сравнительный анализ архитектуры, выдвинутой Элом Пруитом, и двух других архитектур, которые они разработали сами. Сравнительную таблицу разослали участникам группы, чтобы они ознакомились со сравнением и подготовились к собранию.

Этап 3. Собрание команды (предварительное обсуждение проектирования)

На собрании Карен и Эд первым делом утвердили решение, согласно которому сначала необходимо определить удельный вес критериев выбора архитектуры. Так как перед собранием все участники получили по электронной почте сообщение, они были в курсе дела. Однако о выборе самого типа архитектуры еще не говорилось.

Команде были предложены архитектуры на выбор; были показаны результаты сравнительного анализа в виде электронной таблицы. После обсуждения каждый из участников внес коррективы в свои оценки, и вся команда подтвердила свое согласие взять за основу уровневую архитектуру, а также использовать образец проектирования State. На собрании прошло детальное обсуждение выбранной архитектуры.

Были высказаны предложения по усовершенствованию архитектуры. Однако Карен решила не учитывать и не ранжировать эти предложения, чтобы не создавать на каждом собрании новую версию архитектуры. Предложения необходимо было обдумать.

Этап 4. Усовершенствование архитектуры

Теперь перед Карен и Эдом стояла задача выполнить декомпозицию каждого из уровней. Для решения этой задачи они поместили два дополнительных архитектурных элемента в отдельные пакеты. На уровне ролевой игры они сформировали пакет для машины состояний под названием *РолеваяИгра*. Для управления персонажами игры, которые перемещаются параллельно, они создали пакет *Персонажи*. Они также создали пакет *ИгроваяСреда*, в котором содержатся классы с описанием зон, по которым будут перемещаться персонажи. И наконец, они предусмотрели возможность создания в перспективе пакета *Артефакты* для описания различных предметов, таких как боевые щиты, мечи, которые будут задействованы в игре. Для этого пакета, отложенного до будущих выпусков, предполагается репозиторная архитектура.

Декомпозиция уровня игры *Встреча* произошла аналогичным же образом, так как многие классы этого уровня наследуются от классов общего уровня игры. Было решено создать ограниченный доступ к пакетам данного уровня, чтобы предотвратить ситуацию, когда любой класс может ссылаться на любой другой.

Ясно, что в скором времени в процессе разработки и сопровождения станет трудно управлять этими ничем не ограниченными ссылками. Чтобы реализовать ограничение доступа к пакетам приложения, решено было использовать образец проектирования Facade. Эд возражал против такого шага по той причине, что это увеличивает объем кода, который необходимо писать программистам и с которым в дальнейшем необходимо будет иметь дело. «Методы будут вызываться не напрямую, — отметил он, — а только через специальные методы объектов Facade, в результате чего возрастает количество методов». В этом случае возникала еще одна проблема: внешние по отношению к пакету объекты не смогут ссылаться даже на внутренние классы пакета (но могут ссылаться на их общие базовые классы). Но Карен убедила Эда, что эту цену можно заплатить за достижение более четкого и ясного интерфейса в каждом пакете.

На следующем собрании команды эти решения по архитектуре были одобрены участниками.

Этап 5. Документирование архитектуры

Чтобы предоставить описание архитектуры в терминах SDD, Карен и Эд использовали разделы 1–5 стандарта IEEE 1016. Поскольку приложение разделено на два уровня, один из которых предназначен для дальнейшего использования в новых ситуациях, то они пришли к решению документировать каркасный уровень ролевой игры отдельным SDD.

Упражнения

Ответы и подсказки для упражнений, помеченных символами «о» или «п», приводятся в конце этой главы.

Вопросы для проверки

П5.1°. В одном-двух предложениях сформулируйте отношения между архитектурой и проектированием.

П5.2°. В одном-двух предложениях сформулируйте отношения между архитектурой и каркасом.

П5.3. В классификации Гарлана и Шоу существуют пять категорий архитектур. Назовите по меньшей мере три из этих категорий.

П5.4°. Некоторые образцы проектирования затрагивают уровень архитектуры. Назовите три-четыре таких модели.

Общие упражнения

О5.1ⁿ. Студенты часто не замечают разницы между диаграммами потоков данных и диаграммами переходов состояний. Предположим, ваше приложение является симулятором, имитирующим работу с клиентами банка в пакетном режиме

(Batch simulation). Сначала устанавливаются характеристики симулятора, затем симулятор выполняется без прерывания. Как можно описать это в терминах приложения, основанного на потоках данных? Для создания диаграммы используйте простую заготовку, состоящую из четырех частей. (Определите эти четыре части. А теперь подумайте, как можно представить данное приложение в виде диаграммы переходов состояний. Какая перспектива описания архитектуры представляется более выгодной?)

О5.2. Какие из образцов проектирования — State, Observer или Intepreter — наилучшим образом подходят для следующих приложений:

- приложение, информирующее клиентов о неожиданных изменениях на фондовом рынке;
- приложение, позволяющее пользователям простым способом задавать стандартные типы писем при создании письма;
- приложение реального времени, которое выдает сведения о техническом состоянии автомобиля.

Упражнения в команде

К5.1. (Архитектура.)

Разработайте архитектуру для своего проекта. Опишите эту архитектуру в терминах стандарта IEEE. За образец можно взять пример, приведенный в данной главе. Ясно укажите, какой тип архитектуры и какие образцы проектирования вы применяете. Покажите по крайней мере еще один вариант архитектуры, который вам кажется подходящим для этого случая. Объясните, почему, по вашему мнению, описанный тип архитектуры может быть рассмотрен как альтернативный. Используйте метрики. Не требуется, чтобы вы выбрали архитектуру автоматически, опираясь только на метрики.

Определите время, потраченное на выполнение данного упражнения. За единицу отсчета времени мы предлагаем выбрать пять минут. Включите в проект электронную таблицу, в которой указывалось бы время, потраченное каждым участником проекта и командой в целом.

Возьмите за образец форму (табл. 5.6), в которой ведется запись времени, потраченного каждым из участников проекта на каждый модуль в отдельности. Сформулируйте свою точку зрения: окупилось ли время, потраченное вами на отслеживание временных затрат, или можно было обойтись без отслеживания.

Таблица 5.6. Форма для учета времени, потраченного на каждый модуль

Участник команды	Модуль			
	1	2	3	4
Смит	10	4		
Джонс		5	12	
Браун	2			14

Критерии оценки.

- Качество архитектуры: «отлично» — архитектура подобрана разумно и имеет разветвленную структуру модулей.
- Качество улучшения собственных навыков: «отлично» — полезная критика, написанная четким языком.

Ответы

П5.1. Архитектура — это часть проекта, а имено элемент высшего уровня.

П5.2. Каркас — это архитектура, которую можно многократно использовать; в каркасе также могут содержаться проект и компоненты реализации.

П5.4. Facade, State, Observer и Interpreter.

Подсказки

О5.1. Диаграмма потоков данных может состоять из следующих элементов обработки данных: *выполнитьСледующееСобытие*, *обновитьПозициюКлиентов*. Соответствующими хранилищами данных могут быть *НамеченныеСобытия*, где будут храниться все события, запланированные на будущее выполнение, и *СтатусКлиентов*, в котором хранится информация о том, какой статус имеет каждый из клиентов. Эти элементы завершают диаграмму потоков данных.

В данной архитектуре могут быть следующие состояния: *Конфигурирование*, *Воспроизведение*, *ОповещениеОРезультатах*.

Примеры

В этом разделе мы приведем описание двух проектов. Первый — это проект каркаса ролевой видеоигры; второй — проект ролевой игры *Встреча*.

Документы SDD для обоих проектов разделены на две части. Первая часть, включающая разделы SDD с 1 по 5, приводятся ниже, и состоят из архитектурных аспектов проектирования. Вторая часть, раздел 6, которая обсуждается в конце главы 6, представляет собой детальное проектирование.

Зависимость игры *Встреча* от каркаса рассматривается в примере игры *Встреча*.

Пример 1. Каркас архитектуры ролевой игры

[Примечание для студентов. Содержание данного раздела приводится в разделе 5.4.3.]

История версий данного документа.

x/yy/zzz К. Петерс: первоначальный черновой эскиз.

x/yy/zzz К. Петерс: усовершенствованная и сильно измененная схема пакета.

x/yy/zzz Э. Браун: критический обзор, замечания.

x/yy/zzz К. Петерс: пересмотрена работа с компонентами, включенными в нее Р. Боствиком.

x/yy/zzz К. Петерс: детали классов перемещены в раздел 3.

1. Введение

1.1. Цель

В данном документе приведено описание пакетов и классов каркаса ролевой видеоигры.

1.2. Описание проекта

Каркас отражает сущность и основы классов ролевой игры. Он создается с учебной целью, чтобы продемонстрировать пример каркаса. Он не предназначен для создания каркаса для коммерческих игр. Он невелик по размерам, что способствует обучению.

1.3. Определения, сокращения и термины

Каркас — собрание взаимосвязанных классов, которые используют для создания семейств приложений посредством наследования или агрегации.

РИ (*ролевая игра*) — видеоигра, в которой персонажи взаимодействуют. Характер взаимодействия зависит от характеристик персонажей и от среды, в которой они находятся.

2. Ссылки

Software Engineering: an Objected-Oriented Perspective. E. Braude, Wiley, 2000.

UML: The Unified Modeling Language User Guide. G. Booch, J. Rumbaugh, I. Jacobson, Addison-Wesley, 1998 [15].

Стандарт IEEE 1016-1987 (утвержденный заново в 1993 году) устанавливает основные направления разработки SDD.

3. Описание декомпозиции

[Примечание для студентов. В данном разделе указывается точно, как следует группировать классы каркаса ролевой видеоигры. Это предполагает декомпозицию на верхнем уровне: детальная декомпозиция, в частности на методы, откладывается до этапа детального проектирования (см. пример в конце главы 6).]

3.1. Модульная декомпозиция

[Примечание для студентов. В данном разделе демонстрируется, как происходит декомпозиция. Разъяснение по каждому этапу приводится в соответствующем подразделе.]

Каркас состоит из пакетов *РолеваяИгра*, *Персонажи*, *Артефакты* и *Схема*. Их декомпозиция показана на рис. 5.36. Классы этих пакетов поясняются далее.

Все классы являются открытыми, если не указано иное. По пометкам на языке UML (выделены на рис. 5.36 курсивом) можно сделать вывод, что все классы каркаса являются абстрактными.

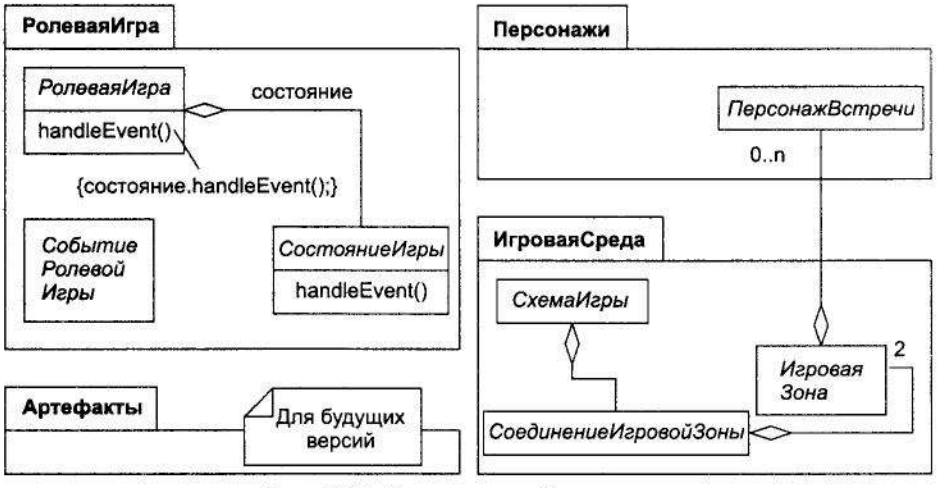


Рис. 5.36. Каркас ролевой видеоигры

3.1.1. Пакет РолеваяИгра

Данный пакет спроектирован как машина переходов по состояниям. Основная идея заключается в том, что ролевая игра всегда находится в одном из нескольких состояний. Данный пакет позволяет описать возможные состояния игры и те действия, которые могут происходить в ответ на события. В пакете реализуется образец проектирования State [Ga]. Состояние игры инкапсулировано в определенный объект *СостояниеИгры*, а он, в свою очередь, агрегируется одиночным объектом *РолеваяИгра*. Этот агрегированный объект называется *состояние*. Другими словами, *состояние* — это атрибут объекта *РолеваяИгра*, относящийся к типу *СостояниеИгры*.

Функция `handleEvent()` класса *РолеваяИгра* вызывается, чтобы управлять каждым событием, возникающим на мониторе (щелчком мыши и т. п.). Она выполняется при вызове функции состояния `handleEvent()`. Применяемая версия функции `handleEvent()` зависит от подкласса *СостояниеИгры*, которому принадлежит свойство *состояние*.

3.1.2. Пакет Персонажи

[Примечание для студентов. Может показаться странным наличие пакета, содержащего всего один класс. Однако в области проектирования программного обеспечения все имеет тенденцию расти. Даже если класс не растет, это еще не говорит о его ненужности. Пример пакета всего с одним классом — java.applet, это пакет с одиночным классом Applet (но в нем содержится также несколько интерфейсов).]

Этот пакет содержит класс *ПерсонажИгры*, в котором дается описание персонажей игры.

3.1.3. Пакет ИгроваяСреда

В данном пакете дается описание физической среды, в которой проходит игра. Класс *СхемаИгры* агрегирует объекты соединений. Каждый объект соединения агрегирует пару объектов *ИгроваяЗона*, которые он связывает. Данная архитектура позволяет осуществлять многочисленные соединения между двумя зонами.

Каждый объект *ИгроваяЗона* агрегирует персонажи игры, которые он содержит (если содержит), и может фиксировать встречу персонажей.

3.1.4. Пакет Артефакты

[Не реализован — для будущих выпусков.]

Данный пакет предназначен для хранения элементов, которые следует расположить в зонах, таких как деревья или столы, а также предметов, которыми владеют персонажи, например щитов и ранцев.

3.2. Декомпозиция на параллельные процессы

Каркас не включает в себя и не затрагивает параллельные процессы.

4. Описание зависимостей

[Примечание для студентов. В данном разделе приводится описание всех зависимостей между модулями.]

Единственная зависимость между модулями каркаса заключается в агрегации класса *ПерсонажИгры* классом *ИгроваяЗона*.

5. Описание интерфейса

Все классы в данных пакетах являются открытыми, таким образом, интерфейсы складываются из всех методов их классов.

Пример 2. Архитектура ролевой игры Встреча. SDD, часть 1

Часть 2 приводится в детальном проектировании примера.

[Примечание для студентов. Содержание данного раздела приводится на рис. 4.2.]

История версий этого документа.

x/yy/zzz К. Петерс: первоначальный эскиз.

x/yy/zzz К. Петерс: составлена общая схема.

x/yy/zzz Э. Браун: выявление дефектов.

x/yy/zzz К. Петерс: дополнительные компоненты, предложенные Э. Брауном.

x/yy/zzz К. Петерс: дополнительная разбивка на компоненты согласно вариантам использования и модели переходов состояний.

1. Введение

1.1. Цель

В данном документе приведено проектирование ролевой видеоигры *Встреча*.

1.2. Описание проекта

Этот проект представляет собой прототип видеоигры *Встреча*, создаваемый с учебной целью, на котором мы продемонстрируем приемы разработки архитектуры, детального проектирования и составления документации. Предполагается использовать выбранную архитектуру в качестве основы для будущих более совершенных версий. В это описание не входят каркасные классы — их разработка документируется в разделе SDD под названием «Каркас архитектуры ролевой игры».

1.3. Определения, сокращения и термины

Нет.

2. Ссылки

Software Engineering: an Objected-Oriented Perspective. E. Braude, Wiley, 2000.

UML: The Unified Modeling Language User Guide. G. Booch, J. Rumbaugh, I. Jacobson, Addison-Wesley, 1998 [15].

Стандарт IEEE 1016-1987 (утвержденный заново в 1993 году) устанавливает основные направления разработки SDD.

3. Описание декомпозиции

Для описания архитектуры *Встречи* используются три модели: вариантов использования, классов и переходов состояний. Кроме того, будут показаны также отношения между каркасными пакетами *Встречи*, описанные в разделе SDD «Каркас архитектуры ролевой игры».

[Примечание для студентов. Мы дополняем стандарт IEEE, вводя в этот документ разделы 3.4. и 3.5. Напомним, что как альтернативу можно рассматривать модель, основанную на потоках данных, однако в нашем случае она не кажется нам особенно полезной. В данном случае для этой видеоигры мы придерживаемся модели переходов состояний как при составлении требований, так и при проектировании.]

3.1. Модульная декомпозиция

[Примечание для студентов. В этом разделе не должно повторяться описание детального проектирования, о котором пойдет речь в следующей главе. Здесь мы не должны углубляться в подробности далее содержимого пакетов.]

Пакеты архитектуры *Встречи* показаны на рис. 5.37. Эти три пакета: *ИграВстреча*, *ПерсонажиВстречи* и *СредаВстречи*. Им соответствуют facade-классы *ИграВстреча*, *РолиВстречи* и *СредаВстречи*. Каждый из этих классов имеет ровно

один экземпляр и по сути является интерфейсом для взаимодействия с пакетом. Остальные классы недоступны за пределами пакета (описание образца проектирования Facade см. в разделе 5.3.2.1 и [33]).

Рис. 5.37. Архитектура и модули ролевой видеоигры Встреча

3.1.1. Пакет ИграВстреча

Пакет *ИграВстреча* состоит из классов, управляющих развитием игры в целом. Этот пакет разрабатывается для того, чтобы обеспечить реакцию на действия пользователя (события).

3.1.2. Пакет ПерсонажиВстречи

Пакет *ПерсонажиВстречи* заключает в себе персонажи, участвующие в игре. Это и персонажи, находящиеся под управленим игрока, и внешние персонажи.

3.1.3. Пакет СредаВстречи

Пакет *СредаВстречи* описывает схему игры *Встреча*, а именно зоны и соединения между ними. К этой категории не относятся какие-либо движущиеся предметы.

3.2. Декомпозиция на параллельные процессы

В игре *Встреча* имеется два параллельных процесса. Первый включает в себя основное действие игры, в котором игрок перемещает свой главный персонаж из одной зоны в другую. Второй процесс состоит из перемещений внешнего персонажа между зонами.

3.3. Декомпозиция данных

[Примечание для студентов. В этом разделе приводится описание структур данных, используемых в приложении.]

Структуры данных, которыми обмениваются пакеты, определяются в классах *Зона*, *ПерсонажВстречи* и *СоединениеЗоныВстречи*.

3.4. Декомпозиция модели переходов состояний

Состояния видеоигры *Встреча* показаны на рис. 5.38.

[Примечание для студентов. Эта диаграмма переходов состояний соответствует диаграмме, используемой в SRS при описании требований. Прочие состояния, упоминаемые в требованиях, будут реализованы в последующих версиях.]

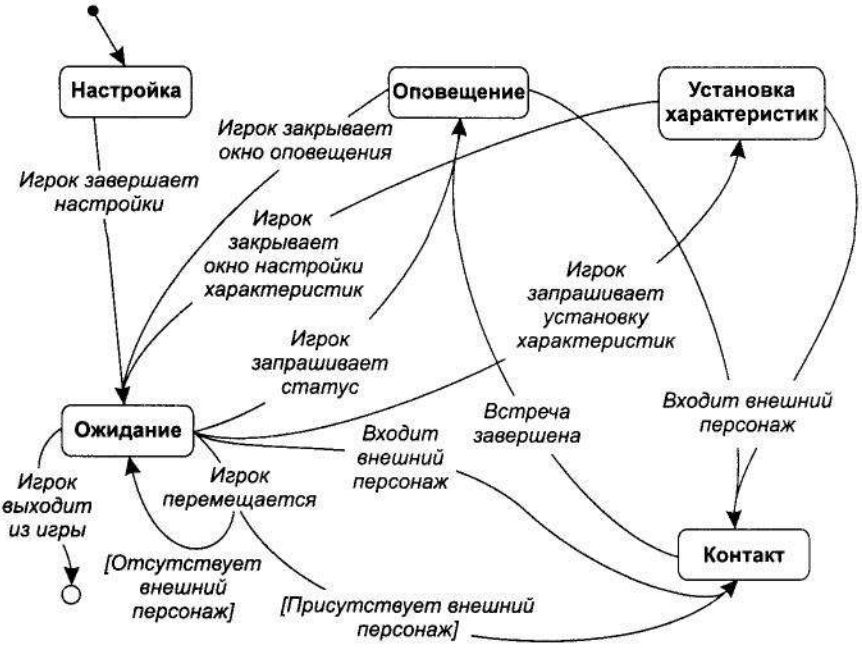


Рис. 5.38. Диаграмма переходов состояний видеоигры Встреча

3.5. Декомпозиция модели вариантов использования

[Примечание для студентов. Этот подраздел мы добавили к IEEE-спецификации, которая не приветствует варианты использования. Поскольку он добавлен в конец раздела, то он никак не нарушает стандартный порядок.]

В видеоигре *Встреча* можно выделить три варианта использования: «Инициализировать», «Перейти в соседнюю зону» и «Вступить в контакт с внешним персонажем» (рис. 5.39). Эти варианты использования подробно описываются в разделе 2.2 SRS, а также в последующих разделах данного документа.

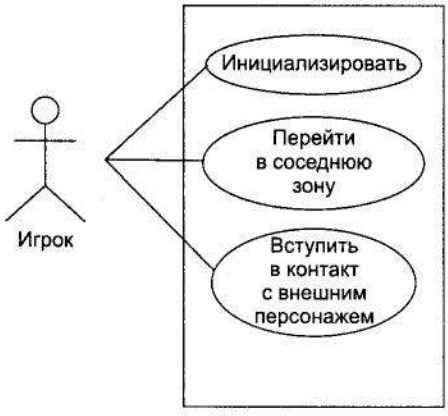


Рис. 5.39. Варианты использования видеоигры Встреча

4. Описание зависимостей

В этом разделе дается описание зависимостей между различными вариантами декомпозиции, перечисленными в разделе 3 данного документа.

[Примечание для студентов. Между вариантами использования нет каких-либо значительных зависимостей.]

4.1. Межмодульные зависимости (объектная модель)

Зависимости между пакетными интерфейсами иллюстрирует рис. 5.40. Пакет *ИграВстреча* является зависимым по отношению ко всем пакетам *Встречи*.

Рис. 5.40. Архитектура видеоигры Встреча

Пакет *СредаВстречи* находится в зависимости от пакета *ПерсонажиВстречи*. Это происходит потому, что любое взаимодействие персонажа игры возможно только в контексте среды. В частности, объекты класса *Зона* отвечают за выявление одновременного присутствия персонажа игрока вместе с внешним персонажем в одной и той же зоне.

Связи между классами, не относящимися к интерфейсу, объясняются далее в этом документе.

4.2. Межпроцессные зависимости

В том случае, когда происходит контакт, во взаимодействие вступают два процесса: процесс перемещения основного персонажа игрока и процесс управления перемещением внешнего персонажа.

4.3. Зависимости внутри данных

Структуры данных, которыми обмениваются пакеты, определены в классах. Взаимодействие классов описано в разделе 6 данного документа.

4.4. Зависимости между состояниями

Каждое состояние зависит от тех состояний, в которые игра может перейти из него.

4.5. Зависимости между уровнями

Зависимость приложения *Встреча* от каркаса ролевой игры показана на рис. 5.41. Каждым пакетом приложения используется именно один каркасный пакет.

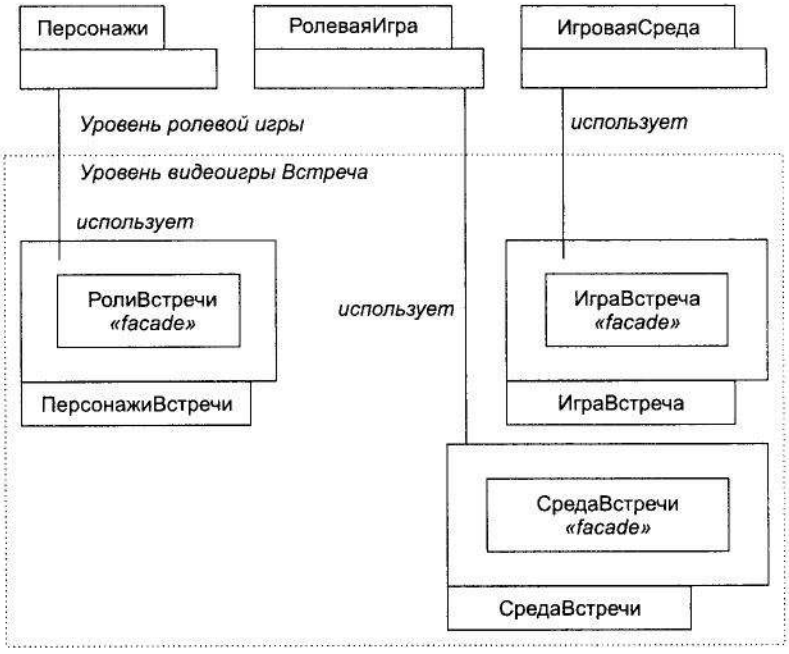


Рис. 5.41. Зависимости между пакетами приложения и каркаса

5. Описание интерфейса

В этом разделе описываются интерфейсы объектной модели. Обратите внимание, что некоторые из описываемых классов определены при описании проектирования каркаса ролевой игры.

5.1. Межмодульные интерфейсы

[Примечание для студентов. В этом подразделе описывается взаимодействие между пакетами.]

5.1.1. Интерфейс пакета ИграВстреча

Интерфейс пакета *ИграВстреча* обеспечивается объектом *играВстреча* класса *ИграВстреча*. Перечислим его состав.

1. `EncounterGame getTheEncounterGame()` // получение единственного экземпляра.
2. `GameState getState()` // текущее состояние экземпляра *ИграВстреча*.
3. `void setState ()` // устанавливает состояние экземпляра *ИграВстреча*.
4. // Любое событие, оказывающее влияние на экземпляр *ИграВстреча*:
   `void handleEvent ( AWTEvent )`

5.1.2. Интерфейс пакета ПерсонажиВстречи

Интерфейс пакета *ПерсонажиВстречи* обеспечивает объект *действующиеЛицаВстречи* класса *РолиВстречи*. Перечислим его состав.

1. `EncounterCast getTheEncounterCast()` // получение одиночного экземпляра.
2. `GameCharacter getThePlayerCharacter()` // уникальный персонаж игрока.
3. `GameCharacter getTheForeignCharacter()` // уникальный внешний персонаж.
4. // Обмен значениями характеристик в зависимости от зоны, в которой происходит контакт:
   `void engagePlayerWithForeignCharacter( GameArea )`

5.1.3. Интерфейс пакета СредаВстречи

Интерфейс пакета *СредаВстречи* обеспечивается объектом *средаВстречи* класса *СредаВстречи*. Перечислим его состав.

1. `EncounterEnvironment getTheEncounterEnvironment()` // получение объекта.
2. `GameArea getArea( String )`
3. `GameAreaConnection getAreaConnection( String )`
4. `void moveForeignChatacterTo( Area ) throws AreaNotAdjacentException`
5. `Image getNeighborhoodAreas( Area )`
   // получение зоны `Area` и зон, удаленных на одно или два соединения.

5.2. Интерфейс процессов

[Примечание для студентов. Мы утверждаем в разделе 3.2, что в игре Встреча имеется два процесса. Это важное для проектирования решение существенно затрагивает интерфейс процесса управления внешним персонажем. Поясним это. Одним из вариантов может быть внешний персонаж в виде потока, который будет управлять собой сам. Такой подход обладает некоторыми преимуществами, но требует, чтобы внешний персонаж либо получал сведения о среде (это может помешать дальнейшему расширению игры), либо был в состоянии определять среду динамически (что было бы изящным решением, но слишком амбициозно для учебного проекта). Архитектура располагает к другому варианту, который и описывается далее.]

5.2.1. Процесс перемещения персонажа игрока

Интерфейс процесса, управляющего перемещениями персонажа игрока, состоит из графического интерфейса пользователя, который описан в SRS. Этот процесс обеспечивает реакцию на события, описанные в разделе 3.4, которые управляются пакетом *ИграВстреча* в соответствии с их спецификацией (см. далее в этом документе).

5.2.2. Процесс перемещения внешнего персонажа

Процесс перемещения внешнего персонажа представляет собой отдельный процесс, находящийся под управлением объекта *ИграВстреча*, с которым он связан. Этот процесс управляется методами, наследуемыми от `java.lang.Thread`.

Глава 6
Детальное проектирование

Реализация и техническая поддержка изящно спроектированных приложений существенно облегчается.

На рис. 6.1 изображена фаза проектирования, о которой пойдет речь в этой главе.

- Основы: разделы 6.1–6.4
- Детали: разделы 6.5–6.10
- Упражнения
- Пример: Проектная документация программного обеспечения (SDD) для видеоигры *Встреча*, часть 2

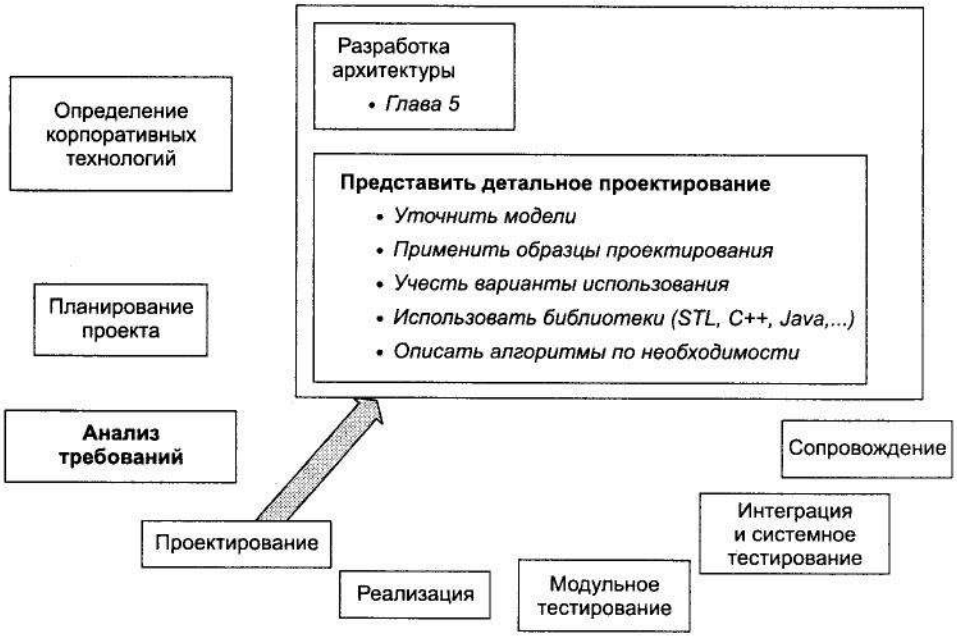


Рис. 6.1. Схема разработки программ: темы главы 6

Учебные цели этой главы состоят в следующем.

- Понять, как образцы проектирования применяются в детальном проектировании.
- Полностью специфицировать классы и функции.
- Определить алгоритмы:
  - используя блок-схемы;
  - используя псевдокод.

Основы

6.1. Введение в детальное проектирование

6.1.1. Что такое детальное проектирование

Детальное проектирование — это техническая деятельность, которая следует за выбором архитектуры. Основной целью этой деятельности является как можно более полная подготовка проекта к его реализации. Другими словами, программисты должны иметь возможность реализовать детальный проект, сконцентрировавшись только на проблемах, связанных с созданием программного кода.

6.1.2. Соотношение вариантов использования, архитектуры и детального проектирования

Взаимосвязь между вариантами использования, архитектурой и детальным проектированием может быть представлена по аналогии с проектированием моста. Варианты использования будут являться частью требований, предъявляемых к конструкции моста. Отталкиваясь от требований, разработчики выбирают архитектуру (например, подвесной мост). После этого они разрабатывают детальный проект для реализации требуемых вариантов использования с учетом выбранной архитектуры (рис. 6.2).

В проектировании программного обеспечения каждый последующий уровень накапливает дополнительные классы. На шаге 1 варианты использования фиксируются как часть требований. На шаге 2 они вместе с другими источниками используются для определения классов предметной области. На шаге 3 мы разрабатываем программную архитектуру (см. главу 5). Последний шаг заключается в проверке того факта, что архитектура и детальный проект удовлетворяют требуемым вариантам использования. Применительно к нашему примеру с проектированием моста мы проверяем то, что автомобили действительно смогут использовать спроектированный мост при поездке с Холма Смита в Долину Джонса, как это было заявлено в вариантах использования. В случае программного проекта мы проверяем то, что классы и методы, специфицированные при детальном проектировании, способны реализовать требуемые варианты использования.

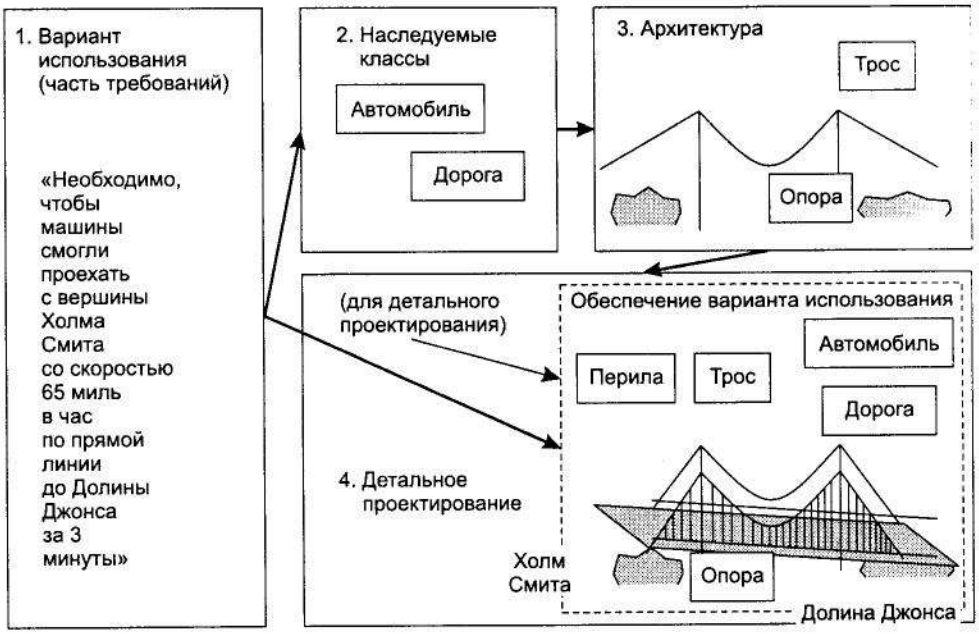


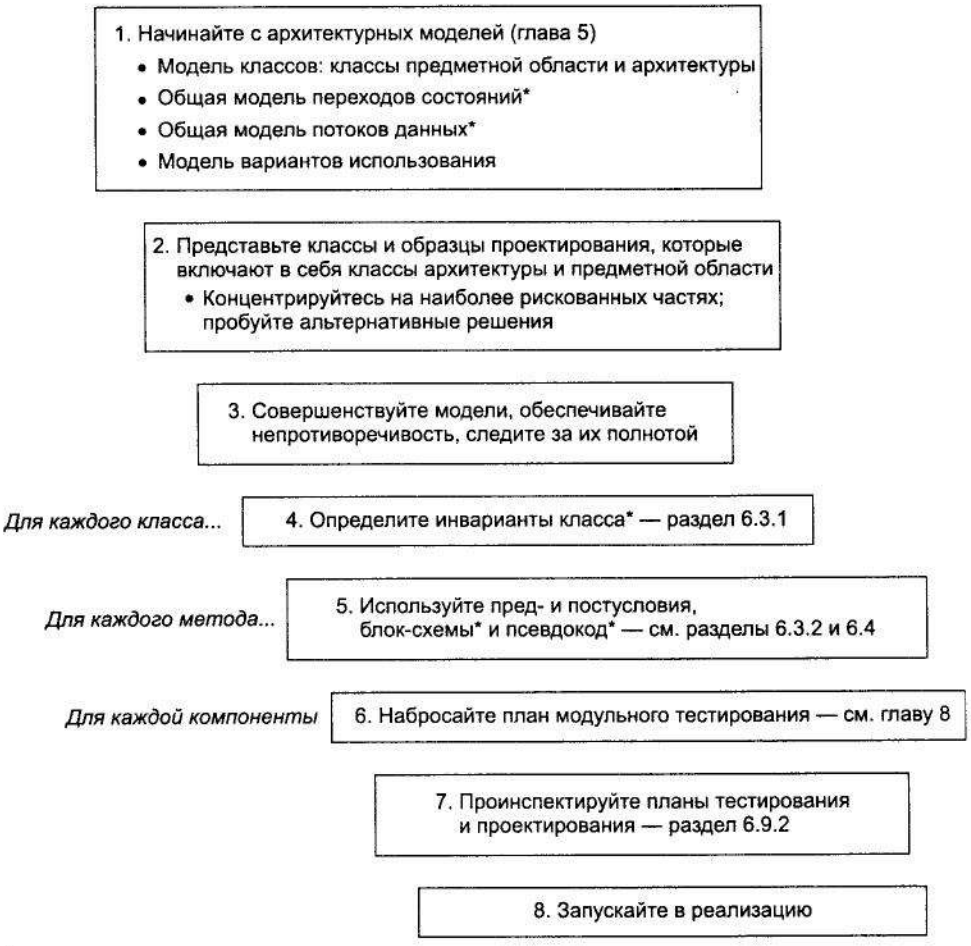
Рис. 6.2. Связь между вариантами использования, архитектурой и детальным проектированием

6.1.3. Типичная схема процесса детального проектирования

Детальное проектирование начинается по результатам фазы архитектуры и завершается с моментом получения полного плана для фазы программирования. Типичная последовательность шагов, предпринимаемых при детальном проектировании, показана на рис. 6.3. В частности, на шаге 2 создаются классы, которые связывают архитектуру с классами предметной области, что проиллюстрировано в предыдущем разделе. При создании таких классов нам могут помочь образцы проектирования. Наиболее разумным было бы начать процесс детального проектирования с аспектов проекта, представляющих наибольший риск. Например, при проектировании игры *Встреча* мы можем посчитать рискованным способ модуляризации классов (все персонажи в одном пакете и т. п.). Мы должны установить это как можно раньше, указав детали методов интерфейса, чтобы иметь детальное представление о том, как организовать эту модуляризацию. Если использование образца проектирования State признано рискованным, значит, нам необходимо сначала специфицировать его детали.

Шаг 3 включает в себя проверку полноты созданного нами проекта. Он также включает в себя проверку того, что объектная модель обеспечивает варианты использования, что обсуждалось в разделе 6.1.2.

Шаг 6 продолжает процесс создания новых тестов по мере появления новых элементов.



*если применимо

Рис. 6.3. Типичная схема детального проектирования

ОДИН ИЗ СПОСОБОВ ОРГАНИЗОВАТЬ КОМАНДУ ДЛЯ ДЕТАЛЬНОГО ПРОЕКТИРОВАНИЯ (1)

1. Подготовьтесь к совещанию по детальному проектированию:
   - убедитесь, что члены команды знакомы с моделями, с которыми им придется работать. Обычно это объектные модели, диаграммы последовательности, переходов состояний или потоков данных;
   - убедитесь, что члены команды знакомы с предлагаемой нотацией. Как правило, это UML со стандартом псевдокода и (или) пример;
   - лидер проекта подготавливает список модулей;
   - лидер проекта создает повестку совещания;
   - лидер проекта выделяет время на обсуждение вопросов повестки (иначе люди могут бесконечно рассуждать о детальном проектировании!); распределите время, выделенное на вопросы повестки совещания.

ОДИН ИЗ СПОСОБОВ ОРГАНИЗОВАТЬ КОМАНДУ ДЛЯ ДЕТАЛЬНОГО ПРОЕКТИРОВАНИЯ (2)

2. Проведите совещание:
   - назначьте кого-то следить за временем обсуждения;
   - убедитесь, что архитектура готова к детальному проектированию. Проверьте, готовы ли интерфейсы модулей. Если не готовы, то проверьте всю группу;
   - не пытайтесь проводить детальное проектирование группой. Это не обязательно; каждый несет личную ответственность. Группы довольно редко выполняют детальное проектирование успешно;
   - выделите модули каждому из членов команды. Запросите оценки времени на проектирование, которые приведут к конкретной дате;
   - записывайте решения и выводы и рассылайте их всем членам команды;
   - решите, как и когда будут обсуждаться результаты.
3. Обновляйте документацию.

   Результатом станет более детальный план-график с модулями и данными инспектирования.
4. Проведите инспектирование детального проектирования.

   См. раздел 6.9.2.
5. Проведите доработку по результатам инспектирования.
6. Проведите последнюю проверку и зафиксируйте полученные выводы.

6.1.4. Проектирование по схеме USDP

В USDP [64] проектирование имеет место в основном во время итераций проектирования и конструирования (рис. 6.4). Фаза анализа зачастую определяется как часть водопадного процесса. В сравнении с терминологией этой книги фаза анализа состоит частично из анализа требований и выбора архитектуры. Якобсон и другие исследователи провели сравнение между процессом анализа и процессом проектирования (табл. 6.1).

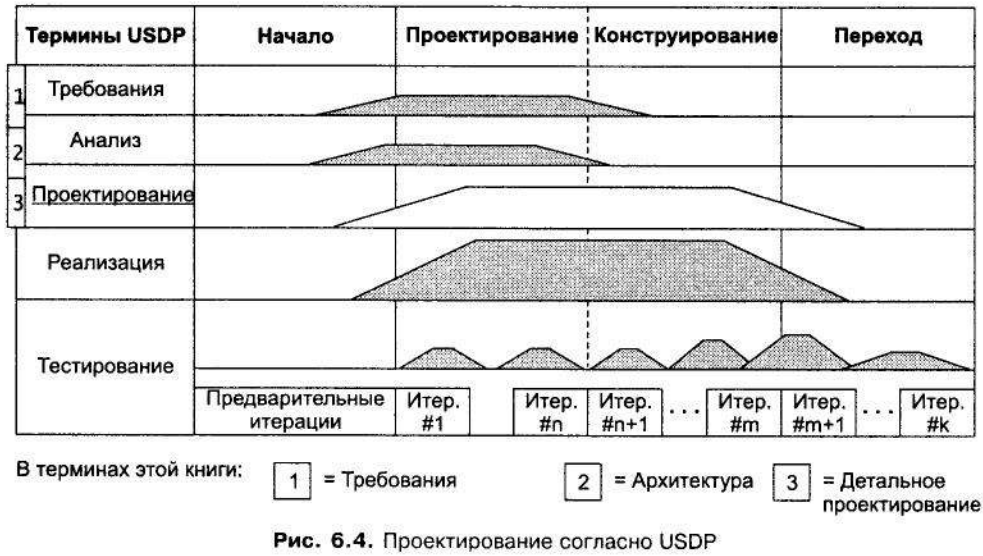


Рис. 6.4. Проектирование согласно USDP

Таблица 6.1. Сравнение анализа и проектирования

Анализ	Проектирование
1. Концептуальный и абстрактный	1. Конкретно: спецификация реализации
2. Применим к различным проектам	2. Зависит от реализации
3. Стереотипы: управление, сущности и границы	3. Нет ограничений на стереотипы классов
4. Менее формален	4. Более формально
5. Менее дорогостоящий в проведении	5. Более дорогостоящее (примерно в 5 раз дороже)
6. Приводит к созданию набросков проекта	6. Провозглашает архитектуру
7. Возникает в результате обдумывания концепции	7. Может использовать инструменты (например, визуальная или циклическая разработка)
8. Информация, поступающая в ходе процесса, менее приоритетна	8. Информация, поступающая в ходе процесса, имеет высокий приоритет
9. Относительно неограничен	9. Ограничено анализом и архитектурой
10. Менее сконцентрирован на диаграммах последовательности	10. Более сфокусировано на последовательности
11. Имеет мало уровней	11. Многоуровневое

Пункт 3 в табл. 6.1 требует отдельного объяснения. USDP поддерживает три типа (стереотипа) классов на уровне анализа, не связанных с классами проектирования: *классы сущности*, *граничные классы* и *классы управления*. В классах сущности выражена суть концепции проекта, и их изменения в приложениях нежелательны. Граничные классы управляют взаимодействием между объектами сущности, а классы управления содержат методы, принадлежащие этим объектам, причем их наличие особенно характерно для приложений, в которых используются классы сущности. Граничные классы обычно аналогичны объекту `Mediator` образца проектирования Mediator (см. раздел 6.5.3.3).

Создатели USDP понимали, что визуальные средства могут быть выгодно использованы в проектировании (см. пункт 7 в табл. 6.1). Примером может служить инструмент Rational Rose (Rational Corporation). Пункт 8 заключает, что важнее поддерживать детальное проектирование на протяжении процесса разработки, чем поддерживать результаты фазы анализа (предполагается, что здесь необходимо сделать выбор). Наконец, в пункте 10 выражен тот факт, что мы возвращаемся к вариантам использования, для того чтобы прийти к согласованности с их диаграммами последовательности.

6.1.5. Проектирование и интерфейсы

В нашем обсуждении различных архитектур в главе 5 мы использовали образец проектирования Facade, благодаря чему смогли получить четкий интерфейс для пакетов. Проектирование с использованием интерфейсов можно сравнить с заключением контракта. Элемент приложения (например, класс *Заказчик*), обеспечивающий требуемую функциональность, гарантирует предоставление функций с определенными именами, типами параметров и возвращаемых значений (например, `void счет( void )` и `boolean печатьСчетов( String типСчета )`). Элементы

приложения, использующие эту основу функциональности, следует проектировать так, чтобы они «не знали» подробностей реализации функциональности.

Применение интерфейсов в проектировании может иметь разные формы. Одна из них — это абстракция. Например, если мы пишем код для объектов *Млекопитающие*, мы должны делать это таким образом, чтобы упоминать только объекты *Животные*. Другими словами, мы пытаемся использовать только интерфейс *Животные*. Это повышает гибкость и применимость нашего проектирования.

Рассмотрим еще один пример. Допустим, мы пишем код приложения для работы с заказчиками. Это можно понимать как написание кода на основе интерфейса *Заказчик*. Вообще говоря, мы можем рассматривать абстрактный класс *Заказчик* с неабстрактным подклассом, например *ПостоянныйЗаказчик* (рис. 6.5). Подобный подход к проектированию более гибок, чем работа с конкретным (не абстрактным) классом *Заказчик*, так он позволяет нам легко добавлять новые типы заказчиков с их индивидуальными версиями метода *счет()*, и при этом нам не нужно менять код, манипулирующий объектами *Заказчик*. Такое деление на абстрактный и конкретный уровни характерно для многих образцов проектирования.

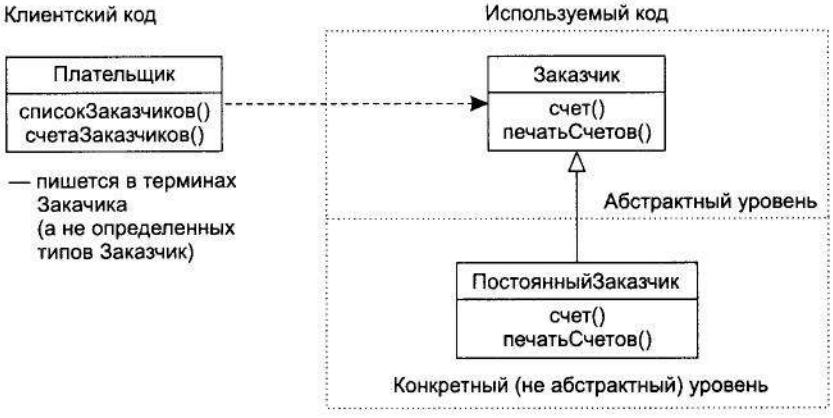


Рис. 6.5. Проектирование и интерфейсы

6.1.6. Повторно используемые компоненты

Большинство инженерных дисциплин (электрика, механика и т. д.) основаны на использовании компонентов, которые могут быть произведены отдельно, независимо от остальных компонентов. Например, проектировщики мостов, вообще говоря, не занимаются проектированием конструкций балок. Из года в год наблюдается одна и та же ситуация: программные разработчики непрерывно «изобретают велосипед», реализуя уже давно существующие процедуры. Одной из причин недостаточного распространения повторного использования программного обеспечения является исторически сложившаяся неудачная организация существующих архитектур, проектов и кода. Развитию повторного использования программного обеспечения поспособствовало широкое распространение

объектно-ориентированных, объектно-подобных и других парадигм компонентов. Благодаря огромному количеству методов, предоставляемых в пакете с каждым классом, необходимая нам функциональность часто уже включена и легкодоступна.

Примером повторного использования программного кода является применение библиотеки Microsoft MFC, элементов управления Visual Basic, объектов COM, Java Beans и других классов Java API. Стандартом для распределенного повторного использования является архитектура CORBA консорциума OMG.

В предыдущей главе, посвященной архитектуре, мы обсуждали каркас приложения. Это пакеты компонентов, спроектированных так, что они допускают повторное использование. Мы разработали каркасы для поддержки выбранной архитектуры приложения, и поэтому их можно использовать повторно достаточно эффективно. Еще одним примером широко применяемого каркаса могут служить базовые пакеты Java API.

Повторно используемые компоненты для Java-приложений обеспечивает стандарт Java Beans. К ним относятся различные графические и «производственные» компоненты, инкапсулирующие корпоративные задачи, такие как доступ к базам данных. Дополнительное преимущество компонентов Java Beans состоит в том, что они подчиняются стандартам, позволяющим им действовать внутри сред разработки.

Веб-приложения (не компоненты), такие как сценарии JavaScript или CGI, также часто бывают повторно используемыми.

Комбинированные возможности по применению стандартных алгоритмов, например сортировки и поиска, могут предоставить библиотеки стандартных шаблонов (STL). STL применяются к различным структурам данных и к объектам практически любых классов. Подробнее речь о них пойдет далее в разделе 6.6.

В целом нужно отметить, что рынок компонентов уже сформировался и постоянно развивается.

Допустим, вы нашли компонент, который, возможно, будет полезен в приложении. Но стоит ли действительно его использовать? Обычно на принятие окончательного решения влияют следующие факторы.

- Документирован ли этот компонент столь же тщательно, что и приложение? Если нет, можно ли это исправить?
- В какой дополнительной настройке нуждаются компонент и (или) приложение?
- Прошел ли компонент тестирование того же уровня (или, что предпочтительнее, еще более жесткое), что и приложение? Если нет, возможно ли это осуществить?

Для сравнительной оценки стоимости различных вариантов можно составить таблицу, аналогичную приведенной в разделе 2.5.2, где мы исследовали, что выгоднее: купить готовое решение или создавать новое приложение.

6.2. Диаграммы последовательности и диаграммы потоков данных в детальном проектировании

Вернемся к диаграммам последовательности (глава 4) и диаграммам потоков данных (глава 3), которые могут быть полезны при описании требований. Чаще всего они используются в качестве инструмента для проектирования. Руководство действиями, которые необходимо предпринять для осуществления детального проектирования, предлагается во врезках «Один из способов», а необходимые подробности и примеры вы найдете далее в тексте.

ОДИН ИЗ СПОСОБОВ УСОВЕРШЕНСТВОВАНИЯ МОДЕЛЕЙ ДЛЯ ДЕТАЛЬНОГО ПРОЕКТИРОВАНИЯ (1). ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТИ

1. Начните с диаграмм последовательности, созданных для детальных требований или архитектуры (если таковые имеются), относящихся к вариантам использования.
2. Если это необходимо, представьте дополнительные варианты использования, для того чтобы описать, как проектируемые части обычно взаимодействуют с остальным приложением.
3. Обеспечьте полную детализацию диаграмм последовательности:
   - убедитесь, что соответствующие объекты и их классы специфицированы;
   - выберите конкретные имена функций вместо естественного языка для представления операций.

ОДИН ИЗ СПОСОБОВ УСОВЕРШЕНСТВОВАНИЯ МОДЕЛЕЙ ДЛЯ ДЕТАЛЬНОГО ПРОЕКТИРОВАНИЯ (2). ДИАГРАММЫ ПОТОКОВ ДАННЫХ

1. Соберите диаграммы потоков данных, созданные для детальных требований и (или) архитектуры (если они имеются).
2. Если это необходимо, представьте дополнительные диаграммы потоков данных, чтобы пояснить потоки данных и процесс.
3. Выясните, каким частям других моделей соответствуют диаграммы потоков данных. Например, «следующая диаграмма потоков данных для каждого объекта *Счет* находится в модели класса».
4. Обеспечивайте полную детализацию диаграммы потоков данных:
   - выясните природу процесса, протекающего в каждом узле;
   - выясните тип передаваемых данных;
   - если описание процесса требует большей детализации, раскройте узлы процесса в диаграмме.

6.2.1. Детальные диаграммы последовательности

Напомним, что варианты использования могут применяться для представления требований, и поэтому мы привлекаем их для определения ключевых наследуемых классов приложения. Теперь возьмем диаграммы последовательности, построенные по вариантам использования, и предоставим соответствующие классы с необходимыми для выполнения последовательностей методами.

Диаграмма последовательности для варианта использования «Встретить внешний персонаж» игры *Встреча*, взятая из примера в конце главы, представлена на рис. 6.6. Вместо словесных описаний функциональности, которую мы использовали для требований (см. раздел 4.4), мы определили собственные функции. Другое усовершенствование заключается в представлении класса *РолиВстречи* образца проектирования Facade, через который должны проходить все внешние ссылки на персонажи игры. Ниже приводятся причины выбора таких функций.

1. Классу *ВнешнийПерсонаж* необходима функция, обеспечивающая отображение на экране. Мы реализуем ее с помощью метода *показать()*. (Поскольку эта возможность необходима для всех персонажей, мы можем гарантировать выполнение данного требования, добавив метод *показать()* в базовый класс *ИграВстреча*.) На диаграмме последовательности показано, что классом *ИграВстреча* создается внешний персонаж (шаг 1.2), затем создается объект *Контакт* и вызывается метод *показать()*.
2. Этот шаг в варианте использования отражает необходимость наличия метода *выполнить()* в классе *Контакт*. На этом шаге требуется, чтобы *Фредди* и *главныйПерсонажИгрока* могли менять значения своих характеристик. Так как эта возможность является общей для всех персонажей *Встречи*, мы заложим ее в базовый класс *ПерсонажВстречи* в методе *установитьХарактеристики()*.
3. На этом шаге необходимо показать результат контакта. Одним из возможных решений является следующее:
   1) сначала *Контакт* создает объект *ОкноКонтакта*;
   2) затем мы выводим окно контакта, вызвав метод *показатьРезультат()*.

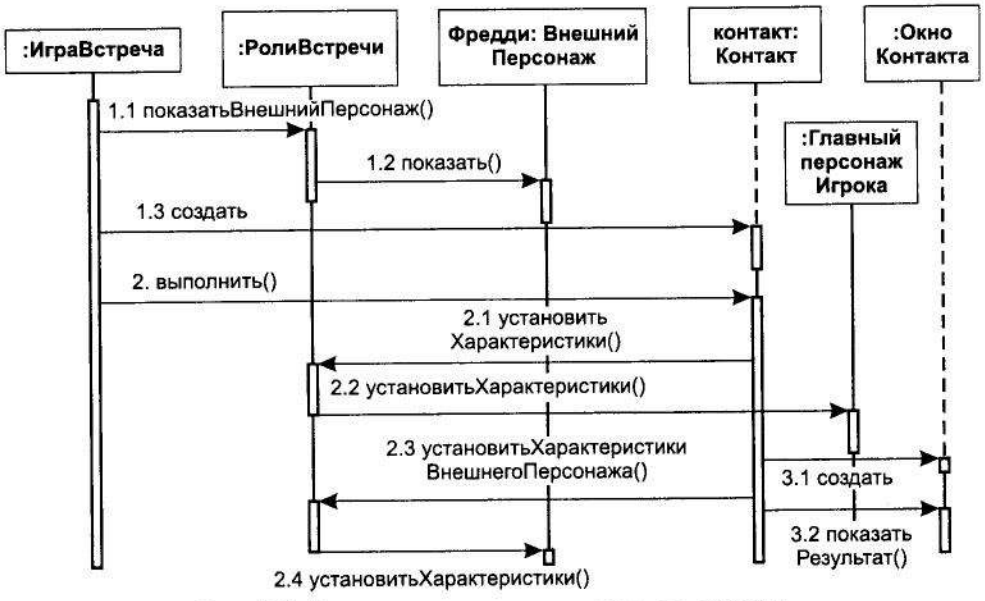


Рис. 6.6. Диаграмма последовательности для варианта использования «Встретить внешний персонаж»

Поскольку все методы, необходимые для реализации этого варианта использования, теперь известны, мы можем представить их на объектной модели (рис. 6.7). Продолжая этот процесс, мы получим детальные модель классов и модель вариантов использования (в виде диаграмм последовательности) (это показано в примере в конце главы). Необходимо также получить детальную модель переходов состояний (если она применима). Диаграмма потоков данных относится к другой модели и обсуждается далее.

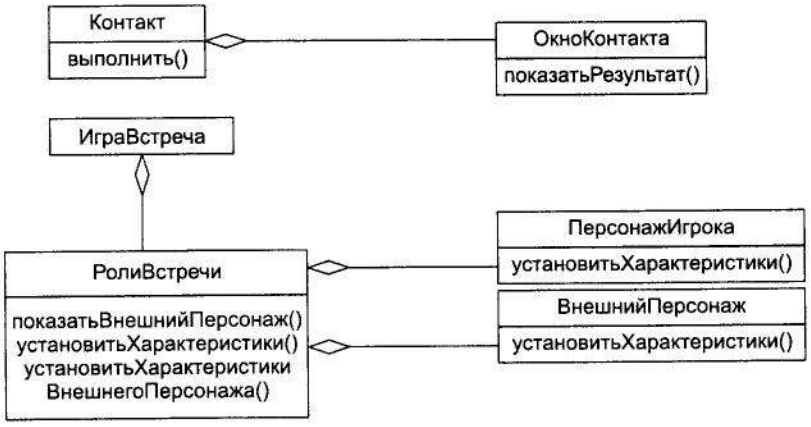


Рис. 6.7. Классы для случая внешнего персонажа игры Встреча

6.2.2. Детальные диаграммы потоков данных

В главе 3 диаграммы потоков данных обсуждались как средство для представления требований. Мы рассмотрели пример с банковским приложением. Для детального проектирования модели данных должны детально описываться, а затем отображаться на функции или классы и методы. Действия, необходимые для описания всех деталей примера банковской диаграммы потоков данных из раздела 5.3.1, демонстрирует рис. 6.8.

Каждый элемент процесса раскрывается в более детальную диаграмму, и этот процесс раскрытия продолжается, пока не будет достигнут самый нижний уровень элементов процесса. На этом уровне элементы, как правило, представляют собой отдельные функции, возможно даже из разных классов. Например, функция `getDeposit()` раскрывается в три функции (получение пароля, его проверка и вывод на экран). Две из этих функций взаимодействуют с хранилищами данных (локальным отчетом транзакций, списком недопустимых пользователей и шаблоном для вывода на экран), которые не были видны на диаграмме верхнего уровня. Отметим, что входные и выходные данные из детальных раскрытий совпадают по версиям с теми данными, из которых они были извлечены.

Не для всех приложений можно извлечь пользу из диаграммы потоков данных. Например, в них нет особой необходимости в нашем примере с игрой *Встреча*.

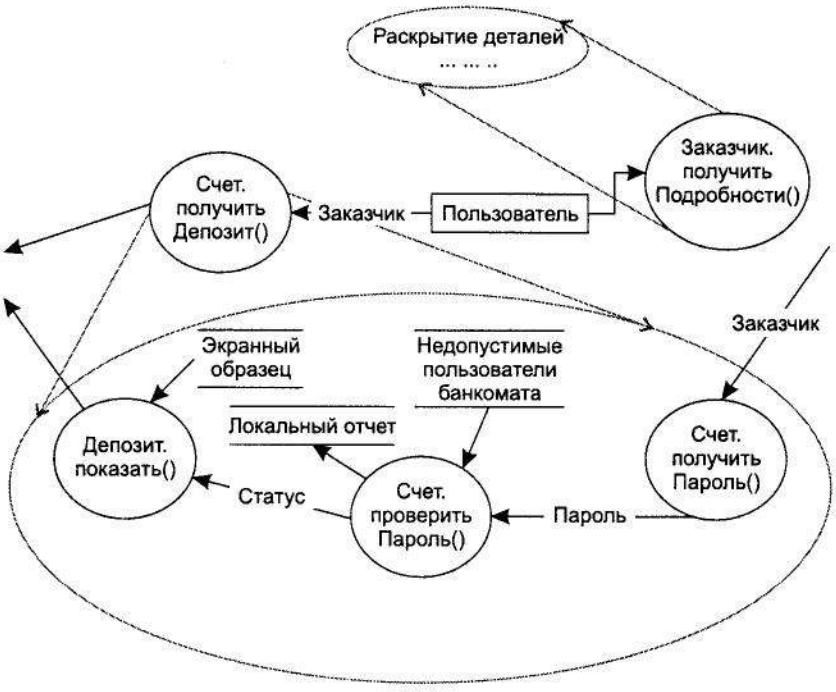


Рис. 6.8. Детальная диаграмма потоков данных для банковского приложения

6.3. Спецификация классов и функций

Целью детального проектирования является полноценный проект, по которому может быть создана программа. Хорошо построенный проект покидает создателя с минимальным числом неопределенностей относительно намерений разработчика. Это справедливо и для программного проектирования. Типичные шаги при осуществлении детального проектирования для каждой функции класса с соответствующими пояснениями представлены ниже.

ОДИН ИЗ СПОСОБОВ СПЕЦИФИЦИРОВАТЬ КЛАСС

1. Соберите атрибуты, перечисленные в SRS. Это проще сделать, если SRS организована по классам.
2. Добавьте дополнительные атрибуты, необходимые для проектирования.
3. Назовите методы, соответствующие каждому из требований для этого класса. Это проще сделать, если SRS организована по классам.
4. Назовите дополнительные методы, необходимые для проектирования.
5. Представьте атрибуты и методы на объектной модели.
6. Установите инварианты класса (раздел 6.3.1).

ОДИН ИЗ СПОСОБОВ СПЕЦИФИЦИРОВАНИЯ ФУНКЦИИ

1. Отметьте разделы SRS или SDD, которым удовлетворяет эта функция (метод).
2. Определите, в каких выражениях функция должна оставаться инвариантной.
3. Задайте предусловия метода (при каких условиях выполняется).
4. Задайте постусловия метода (что происходит в результате его выполнения).
5. Обеспечьте описание используемого алгоритма в виде псевдокода или блок-схемы. Исключением является случай, когда алгоритм тривиален.

Детальные диаграммы классов должны содержать имена всех атрибутов и операций, сигнатуры, область видимости, возвращаемые типы и т. д. Методы средств доступа обычно опускаются. Пример класса детального проекта приложения, контролирующего поток упаковок с дисками на фабрике, производящей микросхемы, представлен на рис. 6.9. В примере используется нотация UML.

Заметим, что специфицированы не все атрибуты. Некоторые обязательные атрибуты могут быть пропущены по решению разработчиков. Также обычно пропускаются различные вспомогательные функции, например `getSize()` и `setSize()`, поскольку их наличие следует из наличия соответствующих атрибутов (`size`).

Инструментарий имеет некоторые преимущества, позволяя разработчикам скрывать (то есть не показывать) определенные элементы, указанные на рис. 6.9. Это может быть, например, раздел «Обязанности» или типы переменных. Кроме того, многие инструменты позволяют разработчикам ограничиться лишь просмотром классов и их взаимосвязей, то есть использовать более общий подход к рассмотрению классов.

Одним из полезных средств создания спецификации классов является CORBA IDL (Interface Definition Language — язык определения интерфейса). Это стандартный текстовый формат для описания интерфейсов, обеспечиваемых наборами классов, их атрибутов и функций. Спецификацию IDL можно найти в [86].

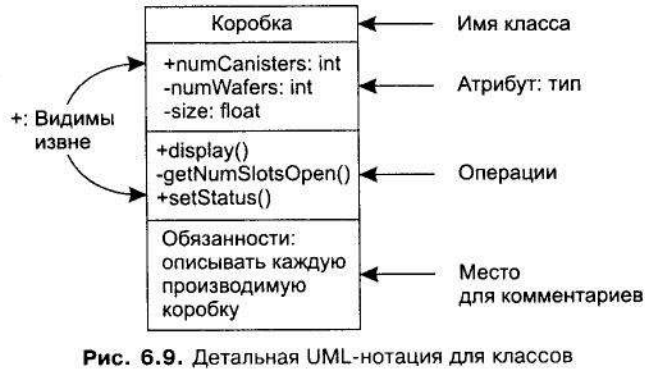


Рис. 6.9. Детальная UML-нотация для классов

6.3.1. Инварианты класса

Инвариант — это утверждение, остающееся справедливым на всем протяжении вычислений. Хорошо продуманные классы часто имеют инварианты. Например,

персонажи видеоигры *Встреча* имеют то свойство, что сумма значений их характеристик всегда является положительным числом.

Инварианты класса представляют собой эффективные требования к объектам класса, выраженные в терминах атрибутов класса. Эти требования могут быть перенесены прямо на требования к приложению, и поэтому могут войти в SRS. В противном случае они предъявляются в ходе проектирования. Инварианты класса принимают форму ограничений на значения. Как и С-требования, они часто называются *правилами бизнеса*. Например, требование «Все участники веб-аукционов должны предоставлять номер своей кредитной карты» может быть переведено в специальный инвариант класса *Участник* следующим образом:

```
зарегистрирован == true AND 400000001 <= номерКредитнойКарты <= 699999999
OR
зарегистрирован == false AND номерКредитнойКарты == 0
```

6.3.2. Инварианты, предусловия и постусловия функций

Один из способов управления работой функций основан на применении инвариантов функций. Так называются утверждения о взаимосвязях переменных, гарантированно удовлетворяемые функциями. Примером из видеоигры *Встреча* может послужить следующий возможный инвариант для метода `adjustQuality()`:

Сумма значений характеристик персонажа постоянна.

Другими словами, когда значение одной из характеристик меняется при помощи метода `adjustQuality()`, значения остальных характеристик изменяются так, что общая сумма значений всех характеристик остается неизменной.

Эффективным способом определения функций является использование предусловий и постусловий. Подробнее об этом рассказывалось в разделе 4.8.4. Предусловия описывают соотношения между переменными и константами, существование которых предполагается до момента выполнения функции. Постусловия описывают эти соотношения после выполнения функции. Например, функция *снятьСоСчета (int сумма)* класса *Счет* может быть описана так:

```
Инвариант снятьСоСчета( сумма )
    доступныеФонды = max (0, баланс)
Предусловие*:
    сумма >= 0 AND
    баланс - сумма >= -МАКСИМАЛЬНОЕ_ПРЕВЫШЕНИЕ_КРЕДИТА
Постусловие*:
    баланс' = баланс - сумма
(x' – это значение x после выполнения функции.)
*Инвариант функции эффективнее дополнительных пред- и постусловия.
```

Заметьте, что банковское определение атрибута *доступные фонды* не включает в себя привилегию превышения кредита заказчиком. Каждый инвариант мо-

жет быть заменен повторением утверждения инварианта одновременно в предусловиях и постусловиях.

6.4. Спецификация алгоритмов

После определения функций, которые необходимо реализовать, может оказаться полезным описать используемый алгоритм, кратко останавливаясь на исходном коде. Основное преимущество этого подхода заключается в том, что разработчик может проинспектировать алгоритм отдельно без углубления в сложности программирования. Благодаря этому некоторые дефекты могут быть обнаружены до того, как они перерастут в дефекты программного кода. Чем важнее метод, тем важнее эта деятельность. Методы со сложным ветвлением являются первыми кандидатами для построения блок-схем.

6.4.1. Блок-схемы

Блок-схемы являются одним из старейших графических представлений алгоритмов. Пример блок-схемы для метода `setName()` класса `GameCharacter` (*ПерсонажИгры*), демонстрирующий два простейших элемента блок-схем — условия (ромбы) и действия (прямоугольники), — приведен на рис. 6.10. Благодаря инспектированию блок-схемы на рис. 6.10 мы можем увидеть, что проверка допустимости имени параметра подчиняется довольно туманной логике.

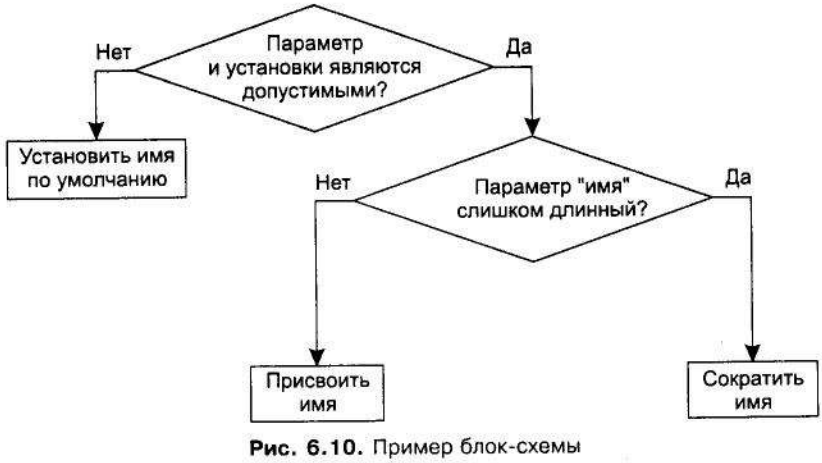


Рис. 6.10. Пример блок-схемы

Весь доступный инструментарий для создания блок-схем из исходного кода — это один из видов обратного проектирования, при котором создаются инспектируемые артефакты, однако допускается инспектирование до программирования, что и является основной идеей использования блок-схем.

Использование блок-схем снизилось за два последних десятилетия двадцатого века. Отчасти это произошло из-за распространения объектно-ориентированного

подхода, который стал довольно популярен именно в те годы в связи с тенденцией сокращения количества ветвлений благодаря распределению функциональности по классам и использованию виртуальных функций.

6.4.1.1. Пример блок-схемы

Следующий пример — алгоритм поиска с возвратами для экспертных систем — иллюстрирует полезность применения блок-схем в объяснении алгоритмов.

Экспертные системы обычно базируются на знаниях, заложенных в виде правил, которые имеют следующий вид:

посылка И посылка И ... И посылка => заключение,

где посылки и заключение являются утверждениями. Например,

животное относится к млекопитающим И животное полосатое => животное является зеброй.

В этом случае наши факты являются обычными строками.

Проблема заключается в создании программы, которая с приведенными ниже исходными данными определяла бы, можно ли сделать вывод, например, о некотором факте L. (Для данного примера ответ «Да».)

1. Список фактов, таких как A, B, Q.
2. Список правил, таких как A & R => L, A & B => C, B & C => F.

Одна из возможных объектных моделей представлена на рис. 6.11. Модель хранит текущий список известных фактов в виде статической структуры. Эта структура хранит список известных правил так же, как статическую структуру. Мы упростим систему этих списков программированием примера, приведенного выше, в терминах классов `Fact` и `Rule`. Основное внимание мы уделим наиболее сложной части: алгоритму `proveBack()`, работающему по принципу поиска от цели. Выясним, может ли некоторый факт, называемый `soughtFact`, быть получен из заданных фактов и правил. Блок-схема этого алгоритма приводится на рис. 6.12. Тщательное инспектирование этой блок-схемы показало бы, что она не в состоянии отслеживать возникновение цикличности в базе правил. Примером может послужить пара правил вида: X => Y и Y => X.

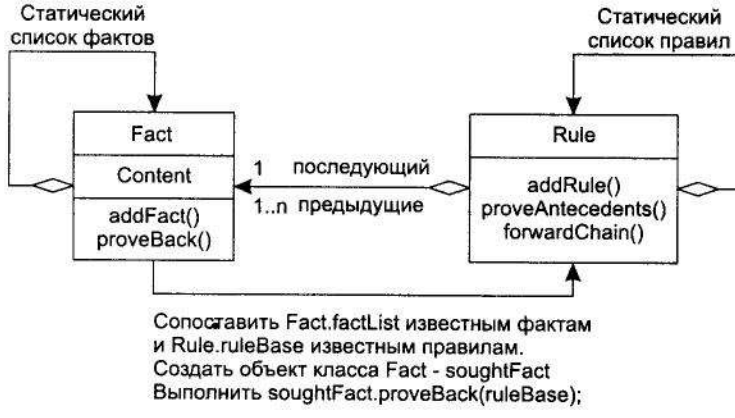


Рис. 6.11. Архитектура для поиска с возвратами

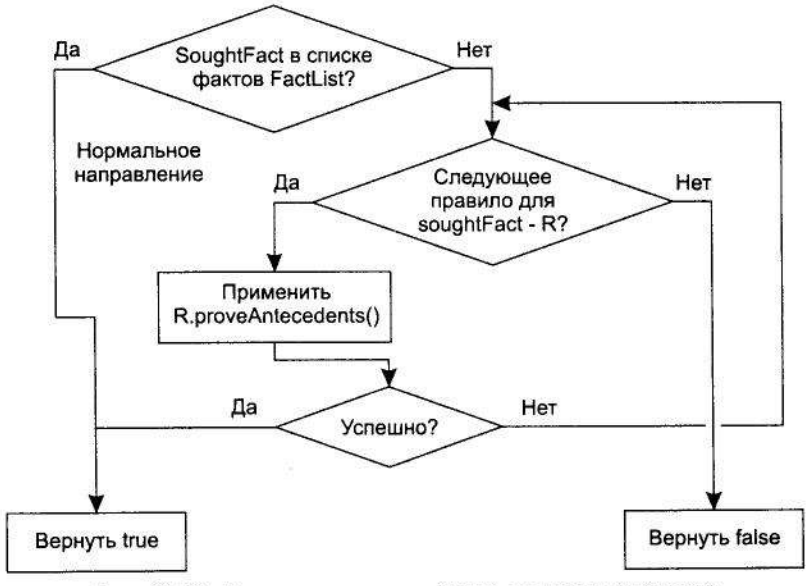


Рис. 6.12. Блок-схема для soughtFact.proveBack (ruleBase)

6.4.2. Псевдокод

Псевдокод — это вариант текстового представления алгоритма, позволяющий не вдаваться в детали языков программирования. В качестве примера приведем псевдокод для гипотетического контроллера рентгеновского аппарата.

```
FOR количество микросекунд, отведенное оператором
    IF количество микросекунд превышает критическое значение
        Попробуйте получить подтверждение у оператора
        IF подтверждение не получено
            Отмена действия с сообщением «отсутствует подтверждение
            оператора на нестандартную длительность» ENDIF ENDIF
    IF уровень напряжения превышает критическое значение
        Отмена действия с сообщением «превышен уровень напряжения» ENDIF
    IF (пациент и экран размещены правильно и
        аппарат прошел самотестирование)
        Включить рентгеновский луч с напряжением р ENDIF… ENDFOR
```

Основным преимуществом псевдокода является его понятность и возможность достаточно точно представить алгоритм. Второе преимущество заключается в возможности инспектирования алгоритмов на предмет их корректности независимо от нагромождений языка программирования. Третье преимущество в том, что мы можем накапливать статистику по проценту ошибок в псевдокоде, а затем использовать эти данные для предсказания количества дефектов в конечном продукте. Многие организации используют инспектированный псевдокод в качестве комментариев в листинге исходного программного кода. Для извлечения

псевдокода из исходного кода существует специальный инструментарий. Например, если выделить псевдокод символами //p, то из следующего программного кода можно получить вышеупомянутый псевдокод:

```
//p FOR количество микросекунд, отведенное оператором
for( int i=0; i<numMicrosecs; ++i )
    //p     IF количество микросекунд превышает критическое значение
    if( numMicrosecs >
        XRayPolicies.CRITICAL_NUM_MICROSECS )
            //p Попробуйте получить подтверждение у оператора
            int supervisorMicrosecsApproval =
                getApprovalOfSuperForLongExposure();
            //p     IF подтверждение не получено
            if( supervisorMicrosecsApproval <= 0 )
                throw( new SupervisorMicrosecsApprovalException() );
```

6.4.3. Когда следует использовать блок-схемы и псевдокод

И псевдокод, и блок-схемы имеют свои преимущества и недостатки. Перечислим их преимущества.

- Делают алгоритм более прозрачным.
- Повышают дисциплину в процессе документирования детального проектирования.
- Дают дополнительный уровень, на котором может быть проведено инспектирование:
  - помогают отследить дефекты до того, как они появляются в программном коде;
  - повышают надежность продукта.
- Часто снижают общую стоимость разработки.

Однако они обладают следующими недостатками.

- Создают дополнительный уровень документации, которую необходимо вести.
- Привносят возможность ошибок при переводе в программный код.
- Могут потребоваться инструменты для извлечения псевдокода и упрощения рисования блок-схем.

Решение об использовании псевдокода и (или) блок-схем принимается в зависимости от каждого конкретного приложения. Некоторые разработчики избегают использования блок-схем, считая их устаревшим средством. Однако блок-схемы и псевдокод могут оказаться очень полезными для отдельной части приложений, где они будут способствовать производству продуктов более высокого качества.

Детали

Эту часть главы можно пропустить при первом чтении и вернуться к ней после прочтения последующих глав.

6.5. Образцы проектирования: приемы детального проектирования

В разделе 5.2.5 мы обсуждали образцы проектирования применительно к архитектуре приложений. Теперь мы будем применять их в детальном проектировании. Само название «образец» предполагает, что это скорее нечто целое, нежели его отдельные части, которое иногда может помочь нам в решении проблем проектирования. Образцы проектирования должны использоваться только в благоприятных условиях, поскольку в противном случае они могут только усложнить обычное проектирование. Несмотря на эти предостережения, автор искренне верит в то, что образцы проектирования являются неотъемлемой частью инструментария программных разработчиков.

В этом разделе мы систематизируем образцы проектирования в соответствии с классификацией, данной в [33]. Ради этой ссылки мы включили образцы проектирования, которые уже обсуждались в предыдущей главе. По этой причине они не будут повторно разъясняться во всех подробностях. Различные веб-источники, например [53], демонстрируют, как образцы проектирования из [33] могут быть реализованы на языках C++ и Java.

Каждый образец проектирования обычно имеет два аспекта. Первый аспект — это обычная модель классов, описывающая классы и их взаимосвязь. Второй аспект представляет способ действия образца проектирования. Для описания последнего может быть применена модель вариантов использования.

ОДИН ИЗ СПОСОБОВ ПРИМЕНЕНИЯ ОБРАЗЦОВ ПРОЕКТИРОВАНИЯ В ДЕТАЛЬНОМ ПРОЕКТИРОВАНИИ

1. Ознакомьтесь с проблемами проектирования, которые можно решить с помощью образцов проектирования.
   - Уясните различия между креационными (К), структурными (С) и поведенческими (П) образцами проектирования.

   Рассмотрите по порядку каждый из этапов детального проектирования.
2. Выясните, к чему относится данная задача: к созданию чего-либо сложного (К), к представлению сложной структуры (С) или она связана с поведением приложения (П).
3. Выясните, существует ли образец проектирования, пригодный для решения этой задачи:
   - используйте введенные обозначения (К), (С) и (П);
   - используйте эту книгу или [33].
4. Решите, что перевешивает — преимущества или недостатки:
   - преимущества обычно включают в себя высокую гибкость;
   - недостатки обычно заключаются в повышенной сложности и низкой эффективности.

В книге [33] образцы проектирования подразделяются на три категории. К первой относятся *креационные* образцы проектирования, отвечающие за процесс создания объектов. Ко второй категории относятся *структурные* образцы проектирования, занимающиеся отображением возможных комбинаций объектов. Третья категория — *поведенческие* образцы проектирования — фиксирует выбранный режим состояния объектов.

6.5.1. Креационные образцы проектирования

Креационные образцы проектирования позволяют разнообразить создание объектов и управлять им, в том числе координировать группы объектов, создаваемые в ходе работы приложения. Без креационных образцов проектирования пришлось бы писать отдельный программный код для каждого типа объектов или группы объектов, которые могут понадобиться в ходе работы приложения. *Креационные образцы проектирования обычно создают объекты при помощи отдельного объекта или метода без прямого использования конструкторов.* Рассмотрим подробнее креационные образцы проектирования, приведенные в [33].

6.5.1.1. Образец проектирования Singleton

Проблема: предположим, что существует единственный объект, принадлежащий классу C. Например, в примере, приведенном в конце главы, у нас имеется только одна видеоигра, таким образом, будет только один объект класса *ИграВстреча*. Причем, как показано в этом примере, различным объектам необходимо ссылаться на этот объект.

Образец проектирования Singleton: идея этого образца проектирования реализуется заданием конструктора скрытого (или, возможно, защищенного) класса. Это предотвращает создание объектов класса `C` за исключением случая их создания одним из методов самого класса `C`. Класс `C` — это заданная часть статических данных типа `C`, которая будет единой. Назовем ее `TheCObject`. Определим открытый метод класса `C`, `getTheObject()`, который будет возвращать объект `TheCObject`, если он уже существует, и создавать его в противном случае. Таким образом, для того, чтобы получить этот единственный элемент класса `C`, мы просто вызываем метод `C.getTheCObject()`.

Статический объект класса `C` `theCObject` может быть инициализирован значением `null`, а затем установлен в значение `new C()` при первом же вызове `getCObject()`. В любом случае нулевое значение `theCObject` может быть использовано для проверки существования этого объекта, упомянутой выше.

В примере в конце главы класс *ИграВстреча* наряду с прочими образцами проектирования использует и Singleton.

6.5.1.2. Образец проектирования Factory

Проблема: необходима возможность написать клиентский программный код, который создавал бы объекты базового класса в ходе работы приложения без использования ссылок на какие-либо подклассы базового класса.

Образец проектирования Factory: комплектует *базовый* класс методом, который создает объекты базового класса. Назовем этот метод `getNewBaseObject()`. Теперь в клиентском коде мы будем использовать `getNewBaseObject()` вместо `newBase()`. Здесь мы использовали свойство виртуальных функций. В ходе работы приложения метод `b.getNewBaseObject()` создает объект подкласса базового класса, которому принадлежит `b`. Использование образца Factory применительно к клиентскому программному коду, ссылающемуся только на базовый класс *БиологическаяКлетка*, демонстрирует рис. 6.13. В ходе работы приложения создается или объект *КлеткаЖивотного*, или объект *КлеткаРастения*, в зависимости от типа класса *БиологическаяКлетка*, на который ссылается программа-клиент.

Рис. 6.13. Пример использования образца проектирования Factory

Мы привели простейший вариант применения образца проектирования Factory.

6.5.1.3. Образец проектирования Prototype

Проблема: предположим, что мы проектируем офисное приложение, которое обрабатывает объекты типа *Документ* и *Заказчик*. Например:

```
Title = d.getTitle(); // d - объект типа Документ
d.printSummary(c);    // c - объект типа Заказчик
d.addToDatabase();
c.addToDatabase();    // и т. д.
```

Предположим, что при каждом запуске наше офисное приложение проделывает эти операции только с одним подтипом объекта *Документ* (например, с объектами типа *ДокументСчет*) и только с одним подтипом объекта *Заказчик*. Приложение определяет эти подтипы непосредственно в ходе работы. Таким образом, во время работы приложения возможны четыре варианта комбинации объектов типа *Документ* и *Заказчик*. Выражения вида:

```
Document d = new Document(); и Customer c = new Customer();
```

позволяют создавать базовые объекты, а не объекты специального типа, необходимого в ходе работы приложения. Это в значительной степени препятствует выполнению кода для всех объектов типа *Документ*.

Образец проектирования Prototype: удачным решением этой проблемы может стать написание приложения с использованием статического объекта типа *Документ* (называемого *прототипом*, который мы назовем `documentPrototypeS`) и объекта типа *Заказчик* (который мы назовем `customerprototypeS`). Кроме того, каждый подкласс класса *Документ* должен иметь метод `clone()`, который создает независимую копию объекта типа *Документ* (аналогично и для объектов типа *Заказчик*). Это показано в объектной модели на рис. 6.14, где *Клиент* — это клиентский класс `MyOfficeApplication`. Программному коду в `MyOfficeApplication` не требуется знать, с каким типом объекта типа *Документ* он имеет дело. Всякий раз, когда методу в `MyOfficeApplication` требуется новый объект типа *Документ*, он выполняет

```
Document d = documentPrototypeS.clone()
```

вместо

```
Document d = new Document()
```

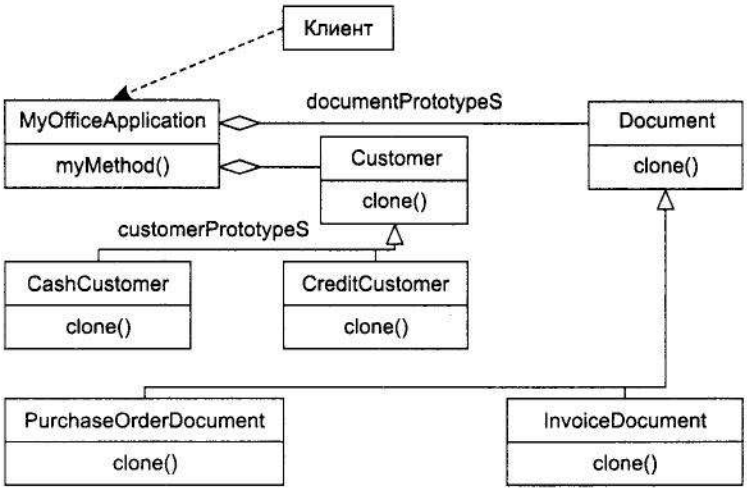


Рис. 6.14. Пример образца проектирования Prototype

Это гарантирует, что во всем приложении будет создаваться согласованный тип объектов *Документ*. Соответствующий программный код представлен в листинге 6.1 и на рис. 6.15.

Листинг 6.1. Образец проектирования Prototype (1)

```
public class MyOfficeApplication
{       private static Document documentPrototypeS;
        private static Customer customerPrototypeS;
```

```
publicMyOfficeApplication
    (Document dPrototypeP. Customer cPrototypeP)
    {   documentPrototypeS = dPrototypeP:
        customerPrototypeS = cPrototypeP:}
    public myMethod
    ...// Необходим новый объект Документ
        Document d = documentPrototypeS.clone();
    ...// Необходим новый объект Заказчик
        Customer c = customerPrototypeS.clone():...
```

Этот класс не знает, для какого типа (подкласса) — Документ или Заказчик — он был вызван.

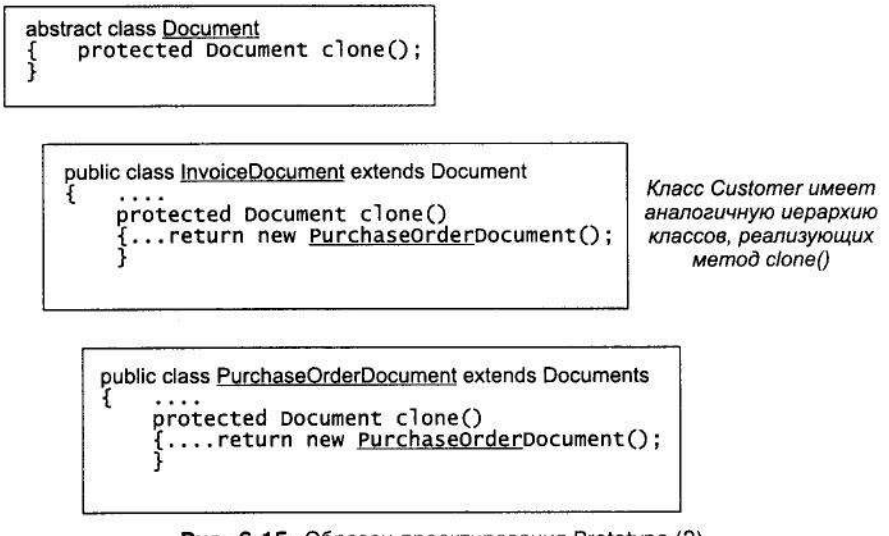


Рис. 6.15. Образец проектирования Prototype (2)

Мы упоминали в разделе 5.2.5, что использование образцов проектирования клиентским кодом, вообще говоря, требует дополнительного наладочного кода. Наладочный код должен быть предоставлен классом `MyOfficeApplication` с прототипом *Документ* (подтипом) и прототипом *Заказчик* в ходе работы приложения с определениями приблизительно следующего вида:

```
m = new MyOfficeApplication // поддерживает прототипы
(new PurchaseOrderDocument(). new CashCustomer() );
```

6.5.1.4. Образцы проектирования Abstract Factory и Builder

Проблема: во время работы приложения необходимо создать совокупность связанных объектов из различных возможных совокупностей. Например, рассмотрим пакет архитектурных программ, который создает планировку домов. Непосредственно во время работы пакета мы хотим увидеть, как одна и та же

планировка дома будет выглядеть в стиле эпохи Тюдоров (где стены, двери, крыша оформлены в соответствующем стиле), в стиле модерн и т. д., просто выбирая стиль, а не перерисовывая весь дом каждый раз при помощи специального программного кода.

Образец проектирования Abstract Factory: Abstract Factory состоит в использовании в клиентском коде метода `AnAbstractFactoryObject.getThing()` вместо `new Thing()`. Во время работы приложения объект `anAbstractFactoryObject` задан как объект необходимого подкласса `AbstactFactory`, благодаря чему согласуются создаваемые им объекты.

Обратимся к примеру в конце главы. Допустим, мы хотим наградить игроков в игре *Встреча*, когда они доберутся до высших уровней, представив игровые зоны и соединения в новом согласованном виде. Зоны с различными стилями оформления на самом деле являются подклассами класса `Area`, а классы соединений являются подклассами `AreaConnection`. Наша задача при проектировании состоит в том, чтобы не допустить ссылок на различные уровни в клиентском коде. Под клиентским кодом здесь понимается код, использующий объекты `Area`. Такое проектирование намного упростит обновление и сопровождение кода.

О неаккуратности и негибкости подхода свидетельствует размещение следующих строк в клиентском коде или в классе `EncounterEnvironment` (СредаВстречи):

```
area[5] = new Area(); // убрать!
connection[5] = new AreaConnection(); // убрать!
```

В таком случае нам было бы неудобно менять уровни зон и уровни соединений во время работы приложения (например, заменять стиль их оформления с уровня 3 на уровень 4). В частности, нам придется заменить эти строки на операторы выбора или на пары операторов вида:

```
area[5] = new Lev8Area(); // неаккуратно
connection[5] = new Lev8AreaConnection(); // неаккуратно
```

Кроме загромождения клиентского кода это также повышает зависимость (связность) между различными частями проектирования.

Мы можем решить эту проблему благодаря использованию класса `Abstract Factory`. Это класс (в нашем случае `EnvironmentFactory`) с абстрактными методами, обещающими создать необходимые компоненты. Каждый из подклассов (в нашем случае: `Level1Factory`, `Level2Factory`, `Level3Factory` и т. д.) имеет согласованные работающие версии этих методов. Взаимосвязи между классами на примере `Level2Factory` продемонстрированы на рис. 6.16.

`EncounterEnvironment` (СредаВстречи) агрегирует объект `EnvironmentFactory`, который мы назвали `envFactory`. Когда пользователь вызывает `getArea()` для объекта `EncounterEnvironment`, этот вызов преобразуется в вызов метода `getArea()` для объекта `envFactory`. Если игра идет на уровне 2, то результатом будет создание зоны уровня 2. Аналогично, вызов `getConnection()` выразится в создании соединений для уровня 2.

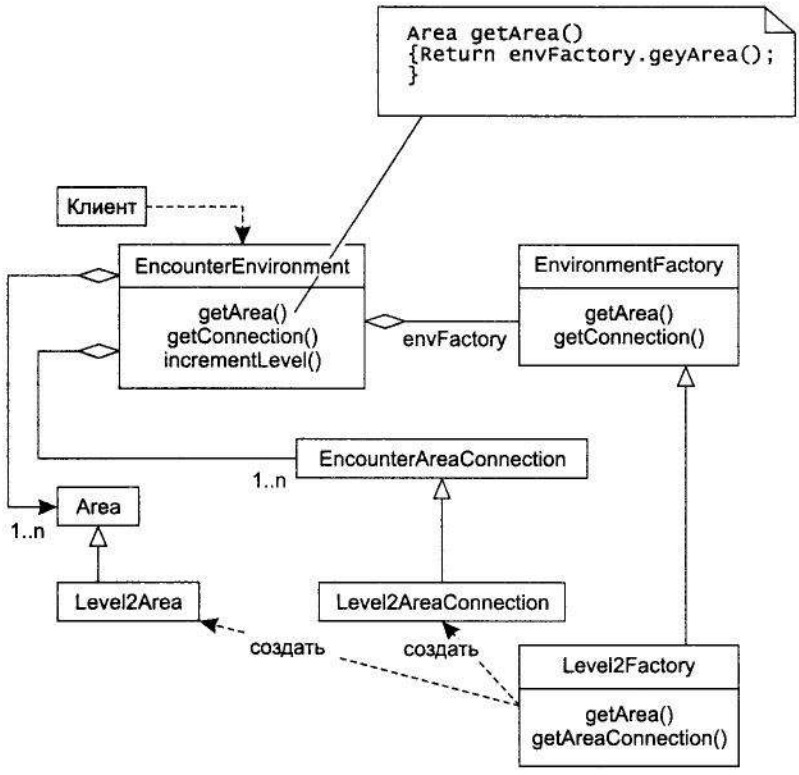


Рис. 6.16. Образец проектирования Abstract Factory, примененный к классу СредаВстречи и демонстрирующий один класс образца проектирования Factory

Пользователь может вызвать `incrementLevel()` для объекта `EncounterEnvironment`, когда игрок достигает следующего уровня. Это приведет к установке `envFactory` в объекте `Level3Builder`. В качестве альтернативы запрос на эти операции может быть передан внешнему управляющему коду (см. раздел 5.2.5). Тот же клиентский код теперь создаст зоны и соединения уровня 3. В клиентском коде используются непосредственно сами уровни.

Объектная модель с тремя уровнями изображена на рис. 6.17. Согласно такой объектной модели, в классе `EncounterEnvironment` вместо строк вида

```
Area[5] = new Area(); // убрать!
```

мы должны использовать

```
area[5] = envFactory.getArea();
```

а вместо

```
connection[5] = new AreaConnection(); //убрать!
```

мы должны использовать

```
connection[5] = envFactory.getAreaConnection()
```

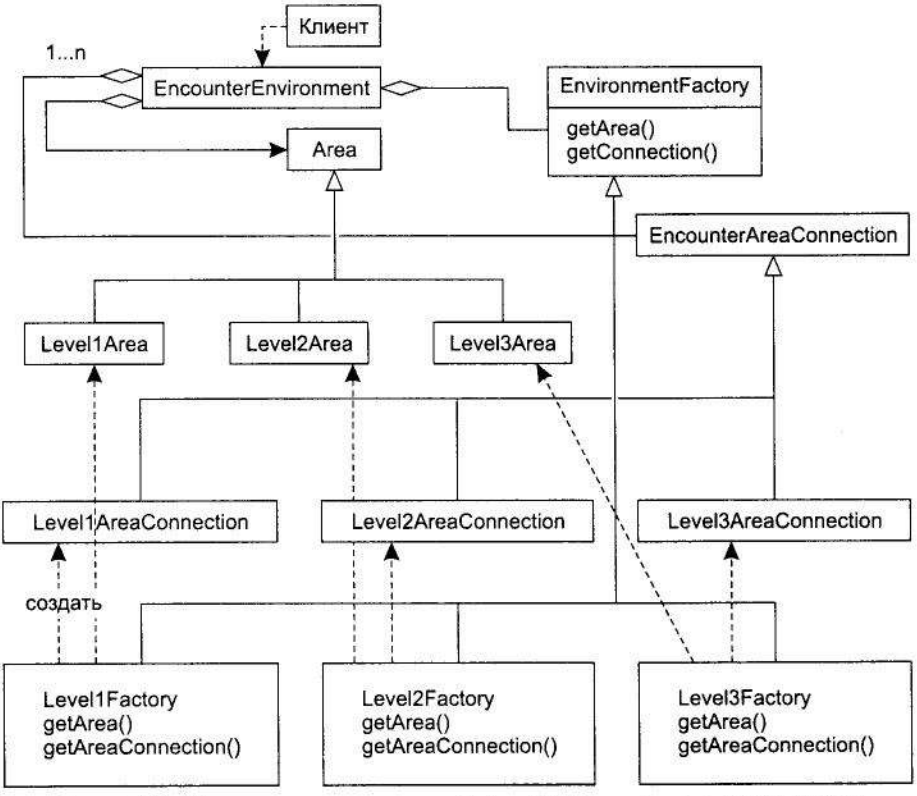


Рис. 6.17. Образец проектирования Abstract Factory, примененный к классу EncounterEnvironment (СредаВстречи) и демонстрирующий несколько классов образца проектирования Factory

После выполнения программы объект `envFactory` представляет собой экземпляр подкласса класса `EnvironmentFactory` (например, это может быть объект `Level9Factory`). В этом случае версия метода `getArea()` для класса `EnvironmentFactory` создает объекты только типа `Level9Area` (это показано пунктирными линиями на рис. 6.17), а версия метода `getAreaConnection()` создает объекты только типа `Level9AreaConnection`. Таким образом, непротиворечивость достигается программным кодом класса `EncounterEnvironment` непосредственно во время работы приложения.

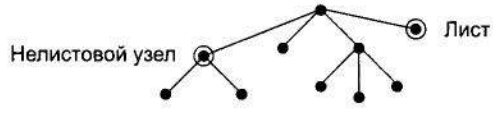
6.5.2. Структурные образцы проектирования

Структурные образцы проектирования представляют комплексные объекты, такие как списки, коллекции и деревья, в виде объектов с удобными интерфейсами.

6.5.2.1. Образцы проектирования Composite и Decorator

Проблема: предположим, мы хотим представить дерево объектов (например, схему организации компании) или связный список объектов.

Образцы проектирования Composite и Decorator: основная идея решения состоит в использовании отношений наследования и агрегации (рис. 6.18). Необходимо показать, что некоторые компоненты агрегируются другими компонентами (рис. 6.19). Модель классов включает в себя компоненты — листья дерева, — которые не являются объединением других компонентов. Образец проектирования действует следующим образом. Объект типа `Composite` выполняет метод `doIt()` либо для листового объекта, что достаточно просто, либо для объекта `NonLeafNode`. Объект `NonLeafNode` вызывает метод `doIt()` на каждого из своих объединенных потомков. Эти объекты типа `Component` продолжают процесс вниз по структуре дерева.



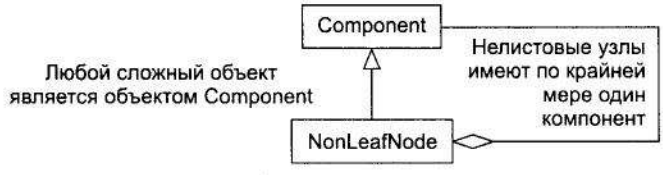


Рис. 6.18. Основа структуры образцов проектирования Composite и Decorator

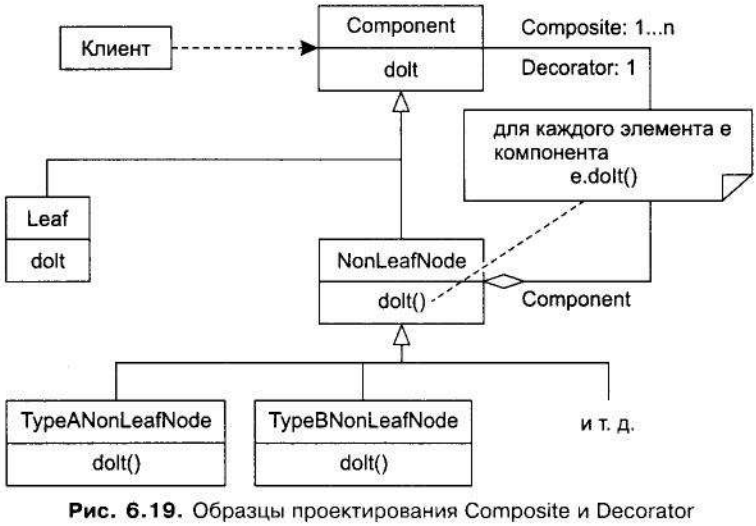


Рис. 6.19. Образцы проектирования Composite и Decorator

Предположим, например, что дерево — это на самом деле схема управления организацией, а метод `doIt()` — это `displayEmployee()`. В этом случае тип объектов `Component` лучше называть `Employee`, а тип `NonLeafNode` — `Supervisor`. Термин «лист» теперь будет соответствовать классу `IndividualContributor`. Вызов метода `displayEmployee()` для объекта типа `Component`, который представляет всю организацию,

приведет к вызову для каждого из объектов своего метода `displayEmployee()`, а затем к вызову `displayEmployee()` для всех его подчиненных объектов. Для вице-президентов (например, `TypeANonLeafNode` бывшего `SeniorVP`) метод `displayEmployee()` может работать по-разному, например включать биографический пресс-релиз, что эквивалентно действию метода `displayEmployee()` для управляющих первого звена (`TypeBNonLeafNode` бывшего `FirstLineSupervisor`), и т. д.

Образец проектирования *Decorator* обладает свойством упрощенной передачи функций. Decorator создает связный список, а объединение (агрегация) (см. рис. 6.18) привлекает только один объект типа `Component`. Например, если `textWindow` является объектом типа `Decorator`, который связан с различными элементами оформления области вокруг блока с текстом (например, полосами прокрутки), то вызов метода `display()` для `textWindow` приведет к тому, что после вывода текста будет выводиться каждый из этих элементов. Текст будет инкапсулирован как объект класса `Leaf`.

6.5.2.2. Образец проектирования Facade

Проблема: приложения могут состоять из сотен классов, тогда как человек в состоянии воспринимать только от пяти до девяти объектов единовременно. Как организовать классы в удобное количество коллекций, каждая из которых имела бы понятный интерфейс?

Образец проектирования Facade: мы уже использовали структурный образец проектирования Facade в главе 5 применительно к архитектуре (см. раздел 5.3.2.1). При использовании образца проектирования Facade становится управляемым конечное количество пакетов (рис. 6.20). Объекты образца проектирования Facade обеспечиваются достаточным количеством открытых методов, для того чтобы позволить пользователям этих пакетов получить доступ ко всем необходимым объектам и требуемой функциональности. Рассуждения в главе 5 акцентированы на выгоде связывания каркасных классов с образцом проектирования Facade.

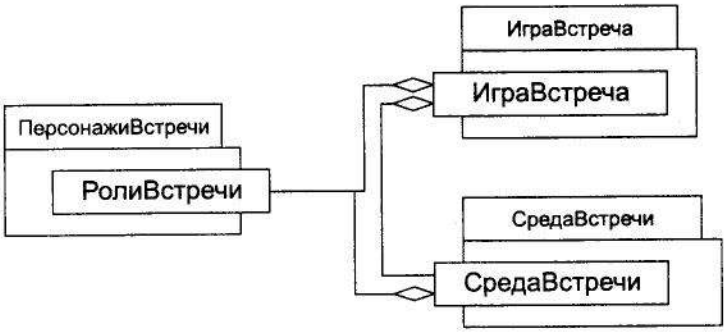


Рис. 6.20. Архитектура игры Встреча: классовая модель

6.5.2.3. Образец проектирования Adapter

Проблема: предположим, что уже существующее приложение или даже просто объект предоставляет функциональность, необходимую нашему приложению. Например, допустим, что приложение вычисляет прибыль от вложения заданной суммы

по заданной процентной ставке на заданное количество лет в некотором виде инвестирования. Мы хотим как можно меньше изменить наше приложение, с тем чтобы использовать преимущество существующего метода вычисления, но при этом быть готовыми к последующей «утилизации» устаревающих методов вычислений.

Образец проектирования Adapter: сначала мы пишем наше приложение, задавая собственные названия функции начисления процентов и классу (объекту), к которому относится эта функция. Например, метод

```
amount( float originalAmount, float numYears, float intRate )
```

относится к классу `Financial`. Этот класс является абстрактным.

Предположим, что функциональность в существующем приложении выражена как метод класса:

```
computeValue( float years, float interest, float amount )
```

Этот метод относится к классу `Principal` (рис. 6.21).

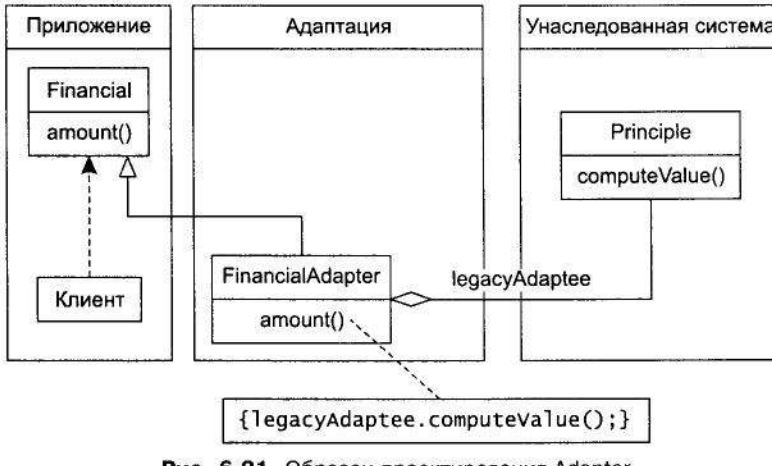


Рис. 6.21. Образец проектирования Adapter

Образец проектирования *Adapter* состоит из класса, который мы для наглядности назовем `FinancialAdapter`. Этот класс унаследован от класса приложения (в нашем случае `Financial`) и агрегирует наследственный класс (в нашем случае `Principal`).

Это можно реализовать, например, на Java следующим образом:

```
Class FinancialAdapter extends Financial {
Principal legacyAdaptee = null;
// Область конструкторов…
/** Этот метод использует наследуемый метод computeValue() */
float amount( float originalAmount, float numYears, float intRate ) {
    return legacyAdaptee.computeValue
        ( originalAmount, numYears, intRate );
    } // конец метода amount()
} // конец класса FinancialAdapter
```

Новое приложение написано применительно к классу Financial и выполняется во время работы относительно объекта FinancialAdapter. (Как показано на объектной модели, объекты FinancialAdapter являются объектами Financial, так что это допустимо.) Например, приложение может быть написано с методом, в котором имеется параметр типа Financial:

```
Void executeFinanceApplication( Financial aFinancial );
```

Этот метод может быть выполнен при следующем вызове:

```
executeFinanceApplication( new FinancialAdapter() );
```

Все вызовы метода amount() класса Financial относятся к наследуемому методу computeValue().

Адаптировать приложение к новой реализации метода amount() в новом классе достаточно легко. Нам потребуется только изменить программный код в классе FinancialAdapter, а оставшаяся часть приложения не будет затронута.

Для сохранения возможности перенастраивания во время работы приложения, мы можем оставить класс FinancialAdapter без изменений, но представить новый класс FinancialAdapter2, наследуемый от Financial. Как только нам потребуется перенастроить приложение на вторую наследуемую систему, нам необходимо будет выполнить приложение при помощи следующего оператора:

```
executeFinanceApplication( new FinancialAdapter2() );
```

Возможность перенастраивания кода внесением только локальных изменений (в противоположность изменениям в нескольких местах) имеет большое значение для разработки и сопровождения.

Путем наследования класса FinancialAdapter такой же эффект может быть получен от классов Financial и Principal вместо использования агрегации.

6.5.2.4. Образец проектирования Flyweight

Проблема: предположим, что приложению придется иметь дело с большим количеством почти неразличимых «мелких» объектов. Например, пусть в нашей видеоигре используются такие артефакты, как *сгустки энергии* и *мечи*. Одновременно могут оказаться активными десятки сгустков энергии (маленьких неразличимых объектов).

Другим примером может послужить приложение, которое представляет текст в различных форматах (печати, просмотра, портативного устройства и т. д.). В принципе, может оказаться полезным определить каждый символ в тексте как отдельный объект. С другой стороны, огромное число различных объектов может обернуться сложностью управления и пустой тратой места для их хранения.

Образец проектирования Flyweight: *Flyweight* предотвращает создание отдельного объекта для каждой элементарной сущности благодаря использованию единственного объекта для всех сущностей и введению параметризованных методов. В вышеприведенном примере мы будем использовать только один объект для всех символов. Альтернативой множеству объектов EnergyPacket (СгустокЭнер-

гии) из видеоигры может послужить единственный объект EnergyPacket с методами, параметризованными в зависимости от контекста, в котором используется этот объект. Игровым персонажам будет необходимо помнить лишь о том, сколькими сгустками энергии они располагают. Таким образом, вместо вызова

```
energyPacket [23].expend() … energyPacket[74].expend ()
```

мы будем использовать

```
theEnergyPacket.expend( главныйПерсонажИгрока. двор ) …
theEnergyPacket.expend( фредди. подвал)
```

Образец проектирования Flyweight обычно используется в связке с объектом Factory, который проверяет, существует ли уже соответствующий объект. Если существует, то возвращается этот объект; иначе объект создается и предоставляется для дальнейшего повторного использования.

6.5.2.5. Образец проектирования Proxy

Проблема: предположим, что запуск метода является достаточно дорогостоящим, поскольку требует значительных временных затрат для такого процесса, как, например, загрузка рисунка из сети или рисование. Возможно, мы будем не в состоянии предотвратить сам вызов метода, но нам хотелось бы предотвратить выполнение его наиболее дорогостоящих частей без особой на то необходимости.

Образец проектирования Proxy: пусть RealActiveClass — это класс, включающий в себя метод expensiveMethod(), который мы не хотим выполнять без особой надобности. Введем абстрактный класс BaseActiveClass, от которого будет унаследован класс RealActiveClass. Образец проектирования *Proxy* предоставляет еще один класс, который мы назовем Proxy. Этот класс также унаследован от класса BaseActiveClass. Пусть класс Proxy содержит метод expensiveMethod(), который ссылается на класс RealActiveClass (рис. 6.22).

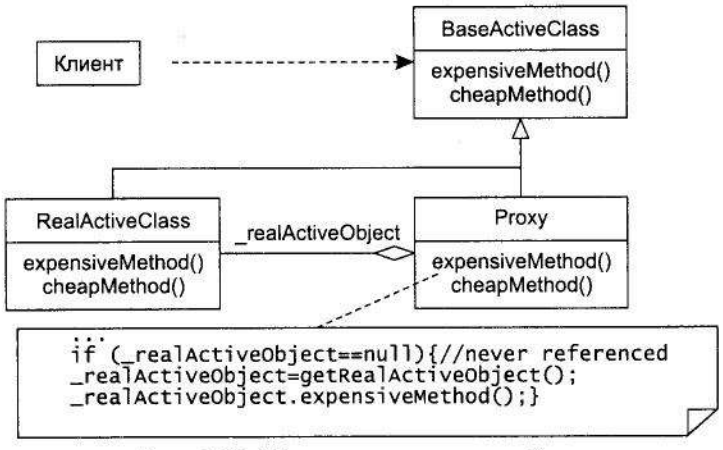


Рис. 6.22. Образец проектирования Proxy

Идея заключается в том, что метод expensiveMethod() класса Proxy является «подделкой» дорогостоящего метода. Клиентское приложение написано в терминах класса BaseActiveClass, а во время работы оно выполняется с объектом типа Proxy. Этот объект перехватывает ссылки, которые шли к классу RealActiveClass. Объект Proxy не позволяет создаваться объектам класса RealActiveClass, за исключением случаев, когда это необходимо. К тому же версия метода expensiveMethod() из класса Proxy, которую вызывает приложение, не выполняет дорогостоящие части более, чем это необходимо. При возникновении такой необходимости объект Proxy передает вызов своего метода expensiveMethod() методу expensiveMethod() объекта RealActiveClass.

Например, RealActiveClass может иметь большой атрибут в виде битовой карты, хранимый на диске. Метод draw() класса Proxy будет извлекать эту битовую карту только в том случае, если она еще не была загружена. Если метод draw() никогда не будет вызван, то битовая карта никогда не будет извлечена с диска. Если это возможно, то при последующем вызове метода draw() битовая карта уже должна быть размещена в памяти резидентно.

6.5.3. Образцы проектирования, основанные на поведении приложения

Образцы проектирования, основанные на поведении, позволяют варьировать поведение приложения в ходе его работы. Без использования этих образцов проектирования нам пришлось бы проектировать и реализовывать каждый вариант поведения по отдельности.

6.5.3.1. Образцы проектирования Interpreter, Observer и State

Образец проектирования Interpreter был описан в разделе 5.3.3. Образец проектирования Observer рассматривался в разделе 5.3.2.2.1. Образец проектирования State был рассмотрен в разделе 5.3.2.3.

6.5.3.2. Образец проектирования Iterator

Проблема: многим приложениям требуется программный код, который «проходил» бы множество объектов различными способами. В качестве примера возьмем внутреннюю структуру организации. Примером применения различных настроек к этому множеству будет процесс распечатки имен сотрудников в алфавитном порядке или по степени их должностной ответственности. Другим примером «прохождения», на этот раз базы данных, является поиск. Например, «Найти всех заказчиков, задолжавших более $500 и опаздывающих с расчетом». Нам необходимо писать этот программный код, концентрируясь на действиях, которые нужно предпринять при обращении к соответствующим объектам, то есть нам требуется возможность описать требуемую последовательность затрагиваемых объектов непосредственно в ходе работы приложения.

Образец проектирования Iterator: образец проектирования *Iterator* базируется на классах `Iterator`, иногда называемых «умными указателями». Каждый класс `Iterator` имеет некоторую определенную структуру. Наиболее эффективной является структура ссылки на элемент множества объектов, для обращения к которым этот класс спроектирован. Наиболее простой пример — это `Iterator`, примененный к массиву. В этом случае в качестве такой структуры (итератора) можно взять просто целое число — индекс элементов в массиве. Работу образца проектирования Iterator демонстрирует рис. 6.23.

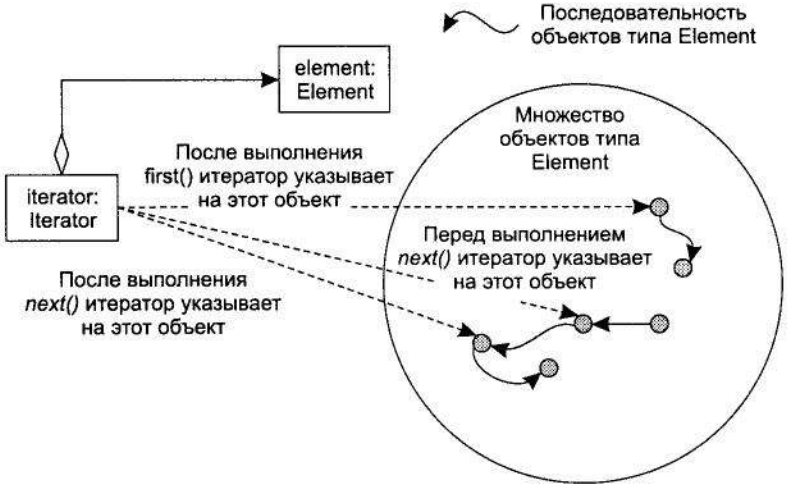


Рис. 6.23. Образец проектирования Iterator

Итераторы предназначены для «посещения» множества объектов класса `Item` при наличии у них следующих методов.

1. `first()` — устанавливает объект `Iterator` как ссылку на *первый* элемент множества.
2. `getItem()` — возвращает ссылку на текущий элемент множества.
3. `next()` — превращает объект (указатель) типа `Iterator` в ссылку, указывающую на *следующий* элемент.
4. `last()` — возвращает `true`, если объект типа `Iterator` указывает за конец множества в результате последовательного прохода по его элементам.

Иногда методы `getItem()` и `next()` объединяются.

Ниже приводится программный код для полного прохождения некоторым итератором `i` множества объектов. Соответствующие элементы, а также их порядок достаточно эффективно описываются итератором `i`. Когда каждый из элементов определен, метод `doWhatever()` выполняет свою функцию над элементом.

```
for( i.first(); i.last(); i.next() )
i.getItem().doWharever();
```

6.5.3.3. Образец проектирования Mediator

Проблема: довольно часто объекты классов тесно взаимодействуют друг с другом, причем достаточно нетривиальными способами. Нам необходимо перехватить поведенческие связи между объектами, не предоставляя объектам сведений друг о друге. Например, в игре *Встреча* при вызове графического интерфейса пользователя для задания характеристик персонажа появляется окно со списком характеристик (Java-объект типа `Component`). Также появляется текстовое окно для задания значения конкретной характеристики игрового персонажа (другой объект типа `Component`). Действия, предпринятые в отношении окна со списком, некоторым образом отражаются и на текстовом окне.

Мы не хотим, чтобы в вывод окна со списком характеристик персонажа был «зашит» расчет значений этих характеристик или способ отображения текстового окна. Это не только может повлиять на переносимость окна со списком, но и будет препятствовать изменениям в проектировании, а также затруднит использование такого же окна со списком в другом выводе при взаимодействии с другими компонентами.

Другой пример взаимодействия объектов — затенение пунктов меню текстового редактора, когда они неприменимы к отображаемому тексту. В данном случае взаимодействуют меню и содержимое экрана.

Образец проектирования Mediator: *Mediator* решает эту проблему встраиванием в каждый промежуточный компонент ссылки на общий промежуточный объект (а не на конкретный промежуточный объект). Этот промежуточный объект и берет на себя бремя технического обеспечения связей между объектами (рис. 6.24).

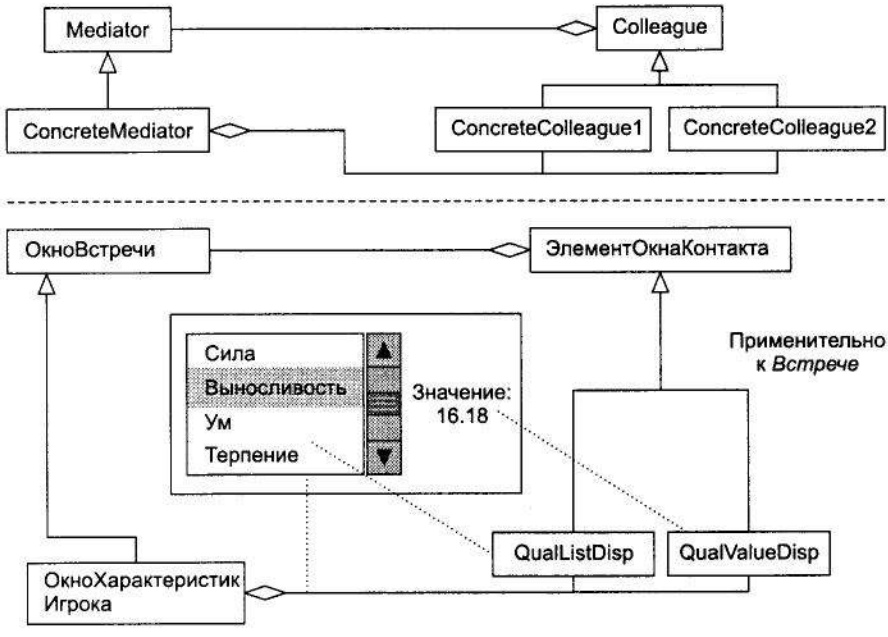


Рис. 6.24. Образец проектирования Mediator

Образец проектирования на рис. 6.24 состоит из базового класса `Mediator`, каждый подкласс которого инкапсулирует необходимую взаимосвязь между объектами, принадлежащими классам `ConcreteColleague1`, `ConcreteColleague2` и т. д. Все взаимодействующие объекты относятся к подклассам базового класса `Colleague`, который ссылается на `Mediator`. Таким образом, взаимодействующим объектам нет необходимости знать друг о друге.

Нижняя часть рис. 6.24 иллюстрирует применение образца проектирования Mediator к примеру с игрой *Встреча*. Когда в окне со списком выбирается одна из характеристик (в левой части рисунка), объект `QualListDisp` сообщает нам, что он агрегировал промежуточный объект `EncounterDisplay` (ОкноВстречи) для обработки этого события. Этот промежуточный объект, заданный непосредственно в ходе работы программы как `SetQualitiesDisplay`, сообщает объекту `QualValueDisp`, что необходимо отобразить значение данной характеристики. Хотя это далеко не прямой путь, однако он приводит нас к достижению цели: `QualListDisp` и `QualValueDisp` не ссылаются друг на друга. Таким образом, эти объекты могут использоваться как составляющая другого поведения приложения (см. пример в конце главы).

6.6. Библиотека стандартных шаблонов (STL) C++

Библиотека стандартных шаблонов (STL — Standard Template Library), созданная А. Степановым, является библиотекой стандартных многократно используемых структур данных и алгоритмов, основанной на использовании шаблонов. STL — это часть стандарта ANSI C++. Исчерпывающую информацию по STL можно найти в [81]. Основные составляющие STL:

- **контейнеры.** Содержит коллекции объектов: векторы, списки, стеки и очереди. Например, для классов `X` класс `list<X>` является контейнером, обеспечивающим доступ к последовательностям объектов переменной длины, который представлен двусвязным списком. `list<X>` имеет метод `push_back(X)` для помещения объектов в конец списка;
- **алгоритмы.** Алгоритмы манипулируют STL-контейнерами и включают в себя: вставку, удаление, поиск и сортировку. Например, алгоритм поиска объекта может быть представлен функцией `find(*X begin, *X end, X objectWanted)`, где `X` является классом;
- **итераторы.** Итераторы представляют операцию над элементами контейнера, что уже обсуждалось в разделе 6.5.3.2;
- **функциональные объекты.** Каждый функциональный объект принимает аргумент и возвращает результат, но при этом он остается исходным объектом. К функциональным объектам относятся следующие операции: деле-

ние, логическое И, «Не равно» и т. п. Одним из примеров является функция сбора выбранных элементов в последовательность:

```
accumulate( InputIterator first, InputIterator last, X initialValue,
BinaryOperation binaryOp )
```

где базовые классы, ссылающиеся на список параметров, определены в STL;

- **адаптеры.** Адаптеры обсуждались выше в разделе 6.5.2.3.

6.7. Стандарты, нотация и инструментальные средства детального проектирования

6.7.1. Стандарт IEEE 890

Вернемся к стандарту IEEE 890 для SDD, описанному в главе 5 применительно к программной архитектуре. Раздел этого документа, посвященный детальному проектированию, состоит из поочередного описания каждого модуля (пакета) с детальным описанием каждого блока данных. Для объектно-ориентированного проектирования последнее может быть заменено детальным описанием каждого класса.

1. **Введение**
   - 1.1. Цель
   - 1.2. Описание проекта
   - 1.3. Определения, сокращения и термины
2. **Ссылки**
3. **Описание декомпозиции**
   - 3.1. Модульная декомпозиция
     - 3.1.1. Описание модуля 1
     - 3.1.2. Описание модуля 2
   - 3.2. Декомпозиция на параллельные процессы
     - 3.2.1. Описание процесса 1
     - 3.2.2. Описание процесса 2
   - 3.3. Декомпозиция данных
     - 3.3.1. Описание блока данных 1
     - 3.3.2. Описание блока данных 2
4. **Описание зависимостей**
   - 4.1. Межмодульные зависимости
   - 4.2. Межпроцессные зависимости
   - 4.3. Зависимости внутри данных

5. **Описание интерфейса**
   5.1. Модульный интерфейс
      5.1.1. Описание модуля 1
      5.1.2. Описание модуля 2
   5.2. Интерфейс процессов
      5.2.1. Описание процесса 1
      5.2.2. Описание процесса 2
6. **Детальное проектирование**
   6.1. Детальное проектирование модулей
      6.1.1. Модуль 1: детали
      6.1.2. Модуль 2: детали
   6.2. Детальное проектирование данных
      6.2.1. Блок данных 1: детали
      6.2.2. Блок данных 2: детали

6.7.2. Язык UML

Мы уже использовали UML для описания вариантов использования и классов, относящихся к архитектуре (разделы 6.3 и 5.2.2). UML позволяет представить все методы в детальном проектировании. Поскольку классы, как правило, содержат множество методов, зачастую желательно их выборочное представление. Это стало возможным благодаря инструментам Rational Rose и Together/J. Они также существенно облегчают масштабирование объектной модели.

6.7.3. Инструменты, использующие исходный код: Javadoc

Диаграммы иерархии классов, использующие гиперссылки, могут быть сгенерированы прямо из исходного кода. Например, Javadoc генерирует списки классов, принадлежащих пакетам, и методов, принадлежащих классам. Из комментариев в исходном коде, оформленных специальным образом, Javadoc также генерирует описания этих компонентов. Исходный программный код с комментариями Javadoc приводится в листинге 6.2. Примеры, в которых широко используется Javadoc, вы найдете в конце глав 7 и 8. Назначение ключевых слов `/**...*/`, `@author`, `@version` и `@param` понятно из приводимого ниже вывода.

Листинг 6.2. Исходный программный код с комментариями Javadoc

```
/**
* Персонаж ролевой игры.
* @author Eric Braude
* @version     0.1. 14.07.98
*/
```

продолжение

Листинг 6.2 (*продолжение*)

```
public abstract class GameCharacter
{
/** Имя игрового персонажа; при инициализации null */
private String _name;

/** Ни одно имя не будет длиннее этой величины  */
public final static int alltimeLimitOfNameLength()
...

/** Для регистрации */
protected void display()
...

/** Доступ к _name */
public String getName()
...

/** Подклассы должны объявлять ограничение на размер имен
*     персонажей
*/
protected abstract int MaxNumCharsInName();

/** Устанавливает значение _name равным aName, если длина
*   не превосходит aMaxNumChars(); в противном случае
*   обрезается.
*   Наследники должны использовать это для setName( String ),
*   но не перезаписывать.
*   @param aName: предлагаемое имя для _name
*   @param aMaxNumChars: ограничение на размер aName
*/
protected final void setName( String aName )
...
```

Из программного кода, приведенного в листинге 6.2, может быть сгенерирована следующая документация Javadoc, представленная ниже в отредактированном виде. Вывод Javadoc можно сделать более привлекательным, если в исходный код вставить теги HTML.

```
Class Characters.GameCharacter
java.lang.Object
|
+ -- Characters.GameCharacter

public abstract class GameCharacter
extends Object
Персонаж ролевой игры.

Variable Index
_name
```

```
    Имя игрового персонажа; при инициализации null
Constructor Index
GameCharacter()
Method Index
alltimeLimitOfNameLength()
    Ни одно имя не будет длиннее этой величины
display()
    Для регистрации
getName()
    Доступ к _name
main(String[ ])
    Протестировать этот класс
maxNumCharsInName()
    Подклассы должны объявлять ограничение на размер имен
setName(String, int)
    Устанавливает значение _name равным aName, если длина
    не превосходит aMaxNumChars(); в противном случае – обрезает.
    Наследники должны использовать это для setName( String ),
    но не перезаписывать
testForGameCharacterClass(String)
    Тестирует все методы данного класса
Variables
_name
private String _name
    Имя игрового персонажа; при инициализации null
Constructors
GameCharacter
public GameCharacter()
Methods
alltimeLimitOfNameLength
    private int alltimeLimitOfNameLength()
    Ни одно имя не будет длиннее этой величины
display
    protected void display()
    Для регистрации
getName
    protected String getName()
    Доступ к _name
maxNumCharsInName
    protected static int maxNumCharsInName()
    Имя игрового персонажа; при инициализации null
setName
    protected final void setName(String aName, int aMaxNumChars)
    Устанавливает значение _name равным aName, если длина
    не превосходит aMaxNumChars(); в противном случае – обрезает.
    Наследники должны использовать это для setName( String ),
    но не перезаписывать
```

```
Parameters:
    aName: предлагаемое имя для _name
    aMaxNumChars: ограничение на размер aName
main
    public static void main(String args[ ])
    Протестировать этот класс
testForGameCharacterClass(String)
    public static void testForGameCharacterClass(String aDestination)
        throws IOException
    Тестирует все методы данного класса
```

Такие инструменты, как Javadoc, реализуют так называемое *обратное проектирование*. Это означает, что вся документация по продукту (включая проект) производится после, а не до создания продукта. В принципе, в любом проектировании в некоторой степени присутствует обратное проектирование. В частности, программисты обычно добавляют методы, не предусмотренные при детальном проектировании. Однако обратное проектирование неэффективно, если оно заменяет выбор архитектуры и детальное проектирование.

Javadoc используется в примере в конце главы. Исчерпывающим источником информации по Javadoc является [67].

6.8. Влияние детального проектирования на проект

В результате проведения детального проектирования план проекта становится более конкретным во многих отношениях. В частности, теперь с большей точностью может быть проведена оценка стоимости проекта, графики работ могут быть разбиты вплоть до конкретных задач, которые, в свою очередь, могут быть распределены между разработчиками.

ОДИН ИЗ СПОСОБОВ МОДЕРНИЗИРОВАНИЯ ПРОЕКТА ПОСЛЕ ЗАВЕРШЕНИЯ ДЕТАЛЬНОГО ПРОЕКТИРОВАНИЯ

1. Убедитесь, что в SDD отражена последняя версия детального проектирования, зафиксированная после инспектирования.
2. Внесите все детали в план-график (SPMP).
3. Поставьте конкретные задачи перед членами команды (SPMP).
4. Уточните оценки стоимости и длительности проекта (см. ниже).
5. Исправьте SCMP, чтобы отразить его новые части.
6. Произведите ревизию процесса, в результате которого было проведено детальное проектирование, и выявите возможные улучшения, включая:
   - затраченное время на подготовку проекта, инспектирование и внесение изменений;
   - результаты поиска дефектов. Определите количество найденных, оставшихся, обнаруженных на стадии детализации, исправленных при детальном проектировании дефектов. Выясните, где они допущены (это относится и к предыдущим фазам, и к стадиям детального проектирования).

6.8.1. Оценка объема работ с помощью детального проектирования

Поскольку с помощью детального проектирования мы можем оценить количество и объем необходимых приложению методов, становится возможной и более точная оценка общего объема работ. Далее (см. главу 2) по объему работ может быть вычислена их стоимость.

ОДИН ИЗ СПОСОБОВ ОЦЕНИТЬ ОБЪЕМ И ДЛИТЕЛЬНОСТЬ ПРОЕКТА ПРИ ПОМОЩИ ДЕТАЛЬНОГО ПРОЕКТИРОВАНИЯ

1. Начните со списка методов.

   Убедитесь в его полноте во избежание получения заниженной оценки.
2. Оцените для каждого метода количество строк программного кода.
   - Классифицируйте методы как очень малые, малые, средние, большие, очень большие.

     Нормальное соотношение методов: ±7%/24%/38%/24%/7%.
   - Используйте свои данные для перевода оценок в количественные значения.

     В качестве альтернативы используйте таблицу Хэмфри (табл. 6.2).
3. Получите общее количество строк программного кода.
4. Преобразуйте количество строк программного кода в человеко-часы.

   По возможности используйте свои методы преобразования.

   В качестве альтернативы используйте опубликованные методы.
5. Убедитесь в том, что ваши оценки размера методов и длительности работ будут проанализированы и сохранены в конце проекта.

Для получения новой оценки продолжительности проекта снова может быть использована модель COCOMO. При определении количества строк программного кода для очень малых, малых и т. д. задач лучше всего использовать личные данные. При отсутствии таких данных могут использоваться корпоративные данные, данные отдела или подразделения. В крайнем случае можно обратиться к таблице Хэмфри [48] (табл. 6.2). Эта таблица помогает определить количество строк программного кода C++, реализующих один метод.

Таблица 6.2. Оценка количества строк программного кода (по Хэмфри)

Тип метода	Категория				
	Очень малые	Малые	Средние	Большие	Очень большие
Вычисления	2,34	5,13	11,25	24,66	54,04
Обработка данных	2,60	4,79	8,84	16,31	30,09
Ввод-вывод	9,01	12,08	16,15	21,62	28,93
Логика	7,55	10,98	15,98	23,25	33,83
Структура	3,88	5,04	6,56	8,53	11,09
Обработка текста	3,75	8,00	17,07	36,41	7,67

Вычислительные методы представляют собой численные расчеты; методы *обработки данных* манипулируют данными (например, изменяют их формат); методы *логики* в основном посвящены ветвлениям; установочные методы выполняют инициализацию; методы *структуры* определяют состояние; методы

обработки текста манипулируют текстом. Оценки методов, совмещающих в себе перечисленные выше типы, могут быть получены усреднением. Например, обычный (средний по объему) метод, который реализует вычисления, однако имеет также значительную логическую составляющую, может быть оценен как имеющий (11,25 + 15,98)/2 = 13,62 строк программного кода.

В [71] приводится оценка, согласно которой в среднем Java и C++ требуют одинакового количества строк программного кода для реализации заданной функциональности.

Такие термины, как «очень малый» и «малый», довольно расплывчаты, поскольку выражают лишь количественные категории, а не точные значения. Эти категории могут частично перекрываться, в этом случае следует использовать средние значения из соответствующей таблицы. (Вообще говоря, это является упрощением нечеткости, но вполне удовлетворяет нашим требованиям.) Нечеткие термины в достаточной степени практичны, так как они просты для понимания. К тому же их можно уточнить путем введения категорий с нормальным распределением (рис. 6.25).

Рис. 6.25. Нормальное распределение числа методов соответственно их объему

В среднем около 38 % методов могут быть отнесены к категории средних, 24 % — к малым и 7 % — к очень малым. Эти числа получены на основании того факта, что около 38 % значений в нормальном распределении лежат в пределах половины стандартного отклонения от среднего значения. На практике, если количество методов, которые вы оценили, например, как очень большие, сильно отличается от 7 %, следует смириться с тем, что ваше приложение действительно имеет нестандартное количество очень больших методов, либо провести ревизию ваших оценок.

В качестве примера произведем оценку объема метода `execute()` класса `Engagement` (Контакт). Этот метод реализует пересчет значений характеристик персонажа, что является важнейшим механизмом выполнения процесса контакта. Таким образом, мы можем классифицировать метод `execute()` как «вычисления». Размер метода `execute()` не является чем-то особо выдающимся, поскольку метод состоит из обычного последовательного вычисления значений. Следовательно, мы оценим объем этого метода как «средний». В итоге получим оценку вклада метода `execute()` в общий программный код в размере 11,25 строк.

Организация, соответствующая требованиям пятого уровня модели CMM (см. раздел 1.8.3), вместо действительных размеров методов обычно предоставляет их оценки, с тем чтобы улучшить процесс оценивания.

В примере в конце главы эти оценки применены к видеоигре *Встреча*.

6.9. Детальное проектирование и качество

Качественное выполнение детального проектирования означает, что мы в состоянии его понять. В этом есть и эстетический аспект; кроме того, понятный проект говорит сам за себя. Тем не менее существуют полезные количественные меры для оценки качества проектирования. В этом разделе мы рассмотрим количественные меры эффективности детального проектирования. Шаги, гарантирующие качество детального проектирования, демонстрируются врезками «Один из способов...».

ОДИН ИЗ СПОСОБОВ ПРОВЕСТИ ИНСПЕКТИРОВАНИЕ ДЕТАЛЬНОГО ПРОЕКТИРОВАНИЯ (1)

1. Будьте готовы фиксировать метрики во время процесса проектирования.
   - В том числе затрачиваемое время, типы дефектов и строгость требований.
2. Убедитесь, что каждый модуль архитектуры полностью раскрыт.
3. Убедитесь, что каждая деталь относится непосредственно к архитектуре.
   - Если какая-либо деталь не принадлежит ни к одному из модулей архитектуры, возможно, потребуется пересмотреть архитектуру.
4. Убедитесь, что вариант проектирования охватывает все необходимые функции.
5. Убедитесь, что проектирование является полным (в смысле классов и методов).
6. Убедитесь в том, что проектный вариант тестируем.

Вопросы проведения инспектирования обсуждались в главе 1.

ОДИН ИЗ СПОСОБОВ ПРОВЕСТИ ИНСПЕКТИРОВАНИЕ ДЕТАЛЬНОГО ПРОЕКТИРОВАНИЯ (2)

1. Проверьте детальное проектирование на:
   - простоту.

     Проектный вариант, который немногие могут понять (даже после серьезных усилий!) достаточно дорог для технической поддержки и может привести к появлению дефектов;
   - обобщенность.

     Допускается ли возможность проектирования аналогичных приложений;
   - расширяемость.

     Допускаются ли улучшения;
   - эффективность.

     Скорость, хранение данных;
   - переносимость.
2. Убедитесь в том, что обеспечена полная детализация:
   - только программный код не является «деталью»;
   - детальная работа все равно должна быть когда-то сделана, и это лучшее время для ее выполнения: не откладывайте!

Некоторые метрики для детального проектирования, соответствующие шагу 1 «Один из способов провести инспектирование детального проектирования», представлены в этом разделе. Шаги 2, 3 и 4 проверяют то, что детальное проектирование охватывает всю архитектуру, и только ее.

Шаг 5 гарантирует полноту проектирования. То, что каждый метод каждого класса правильно описан, проверить достаточно легко, но как мы можем удостовериться в том, что предусмотрели все необходимые методы и классы? Для этого необходимо вернуться к требованиям и убедиться, что разработанный нами детальный проект отвечает всем поставленным требованиям. Если мы имеем структуру организации требований, подобную приведенной в примере в конце главы, то мы точно знаем, что каждому функциональному требованию соответствует свой метод. Таким образом, проверка полноты функциональности сводится к гарантии того, что каждый из методов будет вызван в соответствующий момент работы приложения. Например, рассмотрим следующее требование.

3.2.ПВ.3.2. Возможность настройки характеристик персонажа

Значение любой характеристики игрового персонажа может быть задано, как только персонаж оказывается в одиночестве в игровой зоне. Задаваемое значение должно быть меньше либо равно сумме значений всех характеристик персонажа.

Мы уже убедились в существовании функции, реализующей это требование. Однако для того чтобы убедиться, что наш проект обеспечивает выполнение этой функции, нам необходимо «пройти» через всю последовательность возможных вызовов этой функции. Для этого необходимо разработать набор тестов, результаты которых следует сохранить для фазы тестирования. Вот один из примеров.

Начать игру; вызвать окно для задания характеристик; задать характеристику; еще раз задать характеристику; закрыть окно.

Попасть в зону без внешних персонажей; вызвать окно для задания характеристик; задать характеристику; закрыть окно.

Закончить контакт; дождаться ухода внешнего персонажа; вызвать окно для задания характеристик; задать характеристику; закрыть окно.

Для каждого из этих сценариев мы проверили то, что соответствующие классы и методы действительно существуют. Как только мы сделали это для каждого функционального требования, можно переходить к проверке нашего проекта с точки зрения функциональности. Аналогично мы можем поступить с нашим детальным проектированием для каждого нефункционального требования: мы можем проверить мысленно или с помощью вычислений (например, рассчитывая время), обеспечивается ли нашим проектированием каждое из этих требований. И снова работа по созданию таких последовательностей может быть использована позднее при разработке тестов. Затем мы продолжаем выполнять шаги, представленные в «Одном из способов провести инспектирование детального проектирования» (1) и (2).

Шаг 6 осуществляется для обеспечения возможности тестирования. Другими словами, на этом шаге мы выясняем, удобны ли элементы проектирования для тестирования. Проект, который не может быть легко разделен на составные части, скорее всего, будет невозможно тестировать. Эффективным способом проверки этого свойства является написание тестов для каждого элемента проекта сразу, как только появляется его спецификация.

Шаг 7 относится к свойствам, которые мы ожидаем от детального проектирования. В идеале мы хотим наличия всех этих свойств, но на практике это зачастую неосуществимо. В частности, простота может препятствовать общности и расширяемости. Например, для получения общности в проектировании на рис. 1.2 мы использовали три класса вместо двух, пойдя на компромисс с простотой. Образцы проектирования тоже довольно часто предоставляют дополнительные классы. Таким образом, лучше заранее определиться с наиболее значимыми свойствами, а затем оценивать проектирование относительно них. Если требуется переносимость, мы можем представить план действий для реализации на любой желаемой платформе.

Шаг 8 проверяет то, что каждая деталь кратко описана до реализации в коде. Обычно разработчики откладывают многие детали до момента реализации в целях экономии времени. Однако зачастую этот подход оказывается ошибочным. Во время реализации необходимо рассматривать множество взаимосвязей, и если заранее продумать все детали и по отдельности проинспектировать их, можно получить существенную отдачу.

6.9.1. Качество и метрики в детальном проектировании

Метрики детального проектирования обычно включают в себя подсчет количества модулей, функций, точек входа и выхода. Для объектно-ориентированных приложений по аналогии подсчитывается количество пакетов, классов, методов, параметров, атрибутов и т. д.

Более общей, а поэтому и более сложной оценкой является метрика 19 стандарта IEEE – *структура проектирования* [59], которая определяет «простоту детального проектирования» программы.

6.9.2. Инспектирование детального проектирования

Основные принципы и правила проведения инспектирования были изложены в разделе 1.6.4. Инспектирование детального проектирования состоит в инспектировании его классов, прототипов методов этих классов (имя, возвращаемый тип, типы параметров), блок-схем и псевдокода, а также взаимоотношений классов и методов в рамках различных моделей. Эти модели могут содержать модели вариантов использования и соответствующие им диаграммы последовательности, модель классов, модели переходов состояний и модели потоков данных.

6.9.2.1. Классификация дефектов

При инспектировании всегда собираются данные о каждом дефекте, такие как его серьезность, тип и возможный источник ошибки в жизненном цикле проекта. Стандарт IEEE 1044.1 классифицирует серьезность дефектов так, как показано в табл. 6.3.

Таблица 6.3. Классификация серьезности ошибок по IEEE 1044.1

Степень серьезности дефекта	Описание
Высочайшая	Сбой вызывает разрушение системы, невосполнимую потерю данных или подвергает опасности персонал
Высокая	Вызывает повреждение критических системных функций при отсутствии обходных путей
Средняя	Вызывает повреждение критических системных функций, однако существует решение этой проблемы
Низкая	Доставляет некоторые неудобства или неприятности
Отсутствует	Ничего из перечисленного

Менее дорогостоящая (ввиду меньших затрат времени на сбор статистики), но и менее полезная классификация может быть получена с помощью использования отбраковки (табл. 6.4).

Таблица 6.4. Классификация серьезности дефектов с использованием отбраковки

Степень серьезности дефекта	Описание
Значительная	Не удовлетворяются требования
Средняя	Дефект не может быть отнесен ни к одному из двух других типов
Незначительная	Дефект, который не повлияет на работу и техническую поддержку приложения

Дефекты могут относиться к одному из перечисленных ниже *типов*, которые взяты из стандарта IEEE 1044.1–1995. Типы, относящиеся к детальному проектированию для инспектирования на уровне Javadoc, помечены символами [XDOC], а для инспектирования на уровне псевдокода – [PS].

- Проблема логики (забытые случаи или шаги; повторенная логика; пропуск граничных условий; неважные функции; некорректное прерывание; недостаток текста условия; проверка неправильных переменных; некорректность итерационных циклов и т. д.) [PS].
- Вычисления (некорректное или недостаточное уравнение; потеря точности; ошибка в знаке) [PS].
- Интерфейс и синхронизация (некорректная обработка прерываний; некорректная синхронизация ввода-вывода, несоответствие модулей или подпрограмм) [PS].
- Обработка данных (некорректная инициализация данных; некорректное получение или хранение данных; некорректный масштаб или единицы измерения данных) [XDOC].
- Границы данных [XDOC, PS].
- Данные (некорректны или отсутствуют данные с датчиков, данные оператора, данные таблиц, внешние данные, выходные данные, входные данные) [XDOC, PS].
- Документация (неоднозначное описание и т. д.) [XDOC, PS].

- Качество документа (несоответствие применяемым стандартам и т. д.) [XDOC, PS].
- Расширение (изменение в требованиях и т. д.) [XDOC, PS].
- Ошибки, возникшие в результате предыдущих исправлений [XDOC, PS].
- Проблема представления (связана с фазой тестирования).
- Возможность взаимодействия (несовместимость с другими программами или компонентами) [XDOC, PS].
- Соответствие стандартам [XDOC, PS].
- Другие (не перечисленные выше) [XDOC, PS].

6.9.2.2. Примеры инспектирования детальных требований

В этом разделе мы будем инспектировать примеры псевдокода. В центре нашего внимания при инспектировании находятся *полноправные* дефекты (когда методы правильно выбраны и соответствуют требованиям) и дефекты *по пропуску* (когда необходимо добавить еще несколько методов).

Псевдокод метода должен быть проверен на удовлетворение соответствующему требованию из SRS или из SDD. Вот, например, набросок D-требований из SRS для игры *Встреча*:

[важно] Все персонажи игры имеют одинаковый набор характеристик. Значение каждой характеристики должно быть неотрицательным вещественным числом, заданным с точностью до одной десятой. Значения всех характеристик изначально заданы одинаковыми, а их сумма составляет 100. В первой версии будут следующие характеристики: сосредоточенность, выносливость, ум, терпение и сила. Значение характеристики не может лежать в промежутке между 0 и 0,5.

Это требование реализовано функцией `adjustQuality( String aQuality, float aQualityValue )` с псевдокодом, который необходимо проинспектировать (рис. 6.26).

Рис. 6.26. Инспектирование псевдокода

Инспектирование этого псевдокода вскрывает следующие дефекты.

1. Строка 9: необходимо упомянуть метод `setQuality()`.
2. Строка 10: недостаточно подробно описано, как получать остальные значения характеристик; кроме того, почему всегда «уменьшить» (почему бы иногда не «увеличить»)?

Напомним, что в процессе инспектирования необходимо лишь указывать на существование дефекта, но ни минуты времени не должно быть потрачено на попытки устранить этот дефект. Классификация дефекта 2 (строка 10) с помощью отбраковки по степени серьезности более предпочтительна, так как ее интерпретация может привести к значительным изменениям в продукте. Используя классификацию стандарта IEEE 1044.1, получим, что это *вычислительный* дефект.

6.10. Подведение итогов

Подведем итоги обсуждения детального проектирования в этой главе.

- Выполняйте детальное проектирование так, чтобы оно позволяло написать программный код.
- Старайтесь использовать стандартные образцы проектирования.
- Определяйте выбранные алгоритмы:
  - с помощью блок-схем;
  - с помощью псевдокода.
- Применяйте выбранный инструментарий.
  - Например, Javadoc.

Упражнения

Ответы и подсказки для упражнений, помеченных символами «о» или «п», приводятся в конце этой главы.

Вопросы для проверки

П6.1. В двух-трех предложениях сформулируйте условия, при которых следует использовать следующие средства детального проектирования:

- блок-схемы;
- псевдокод.

П6.2. Назовите три креационных образца проектирования и сформулируйте их назначение в одном-двух предложениях.

П6.3. Назовите три структурных образца проектирования и сформулируйте их назначение в одном-двух предложениях.

П6.4. Назовите три образца проектирования, основанных на поведении, и сформулируйте их назначение в одном-двух предложениях.

Общие упражнения

О6.1. Выполните оценку числа строк Java-кода для следующего набора методов. Методы грубо упорядочены по возрастанию объема. Считайте, что об этом коде не известно больше ничего.

- Метод 1 — ввод-вывод;
- Метод 2 — обработка текста;
- Метод 3 — вычисления;
- Метод 4 — логика;
- Метод 5 — обработка данных;
- Метод 6 — вычисления;
- Метод 7 — обработка текста;
- Метод 8 — обработка данных;
- Метод 9 — структура;
- Метод 10 — вычисления;
- Метод 11 — обработка текста;
- Метод 12 — обработка данных.

О6.2. Выполните проектирование (как проектирование архитектуры, так и детальное) приложения управления счетом согласно приведенным ниже требованиям. Если вы обнаружите в требованиях дефекты, исправьте их. Составьте отчет о времени, которое потребовалось вам на наиболее важные операции, с точностью до пяти минут.

1. Система отображает текущий баланс в системном окне.
2. Система разрешает пользователю сделать вклад в размере не более $10 000.
3. Система разрешает пользователю снимать со счета любую сумму. Если пользователь попытается снять сумму, превышающую остаток средств на счете, система выводит сообщение: «Эта сумма превышает остаток на счете».

Упражнения в команде

К6.1. (SDD)

Составьте SDD для вашего проекта, опираясь на стандарт IEEE. Используйте при этом самую позднюю версию SRS.

Определите время, потраченное группой и индивидуально. Подсчитайте время для отдельных составляющих работы: архитектура, инспектирование, обзоры,

детальное проектирование. Прокомментируйте полученные результаты и решите, каким образом можно было распределить время более эффективно.

Критерии оценки.

1. Ясность («Отлично» — SDD непосредственно ведет к пониманию архитектуры и детального проектирования).
2. Полнота («Отлично» — достаточно, чтобы охватить все требования).
3. Изящество («Отлично» — допускает расширение и сопровождение).
4. Выражение улучшения процесса («Отлично» — демонстрирует определенное пошаговое улучшение).

Примеры

Мы опишем два варианта детального проектирования: каркаса ролевой видеоигры и приложения *Встреча*. Благодаря такому разделению мы упростим себе повторное использование и сопровождение каркаса.

Пример 1. Каркас ролевой видеоигры. Детальное проектирование (оставшиеся части SDD)

[Примечание для студентов. Это продолжение документа SDD. Его оглавление вы найдете в главе 5.]

6. Детальное проектирование каркаса ролевой видеоигры

Архитектура пакетов, описываемых в этом разделе, приводится на рис. 5.40.

6.1. Детальное проектирование модулей

[Примечание для студентов. В этом разделе приводятся все необходимые нетривиальные свойства модулей, описанных в разделе 3.1 SDD для каркаса ролевой игры.]

6.1.1. Пакет РолеваяИгра

Все события мыши прослушиваются объектами класса *СлушательМышиРолевойИгры*, который наследуется от класса *СлушательМыши* (рис. 6.27).

Объекты, чувствительные к событиям мыши, запрашивают обработку событий у объекта *РолеваяИгра*. Объект *РолеваяИгра* передает управление методу `handleEvent()` агрегированного объекта *СостояниеИгры*. Диаграмма последова-

тельности для этого процесса показана на рис. 6.28. Для данной реализации игры методы либо показаны на рис. 6.28, либо являются тривиальными.

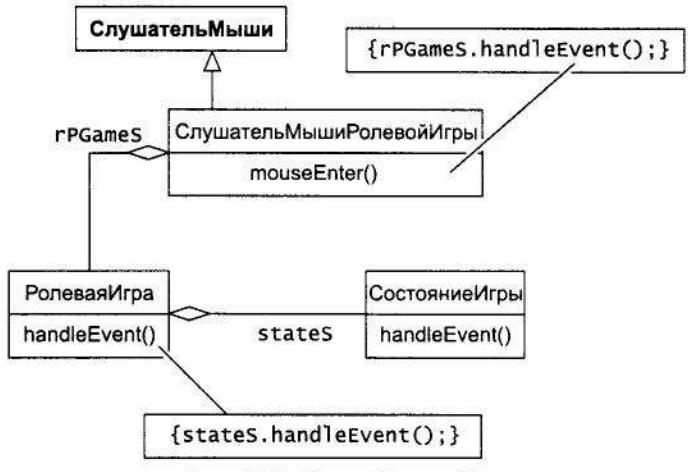


Рис. 6.27. Пакет РолеваяИгра

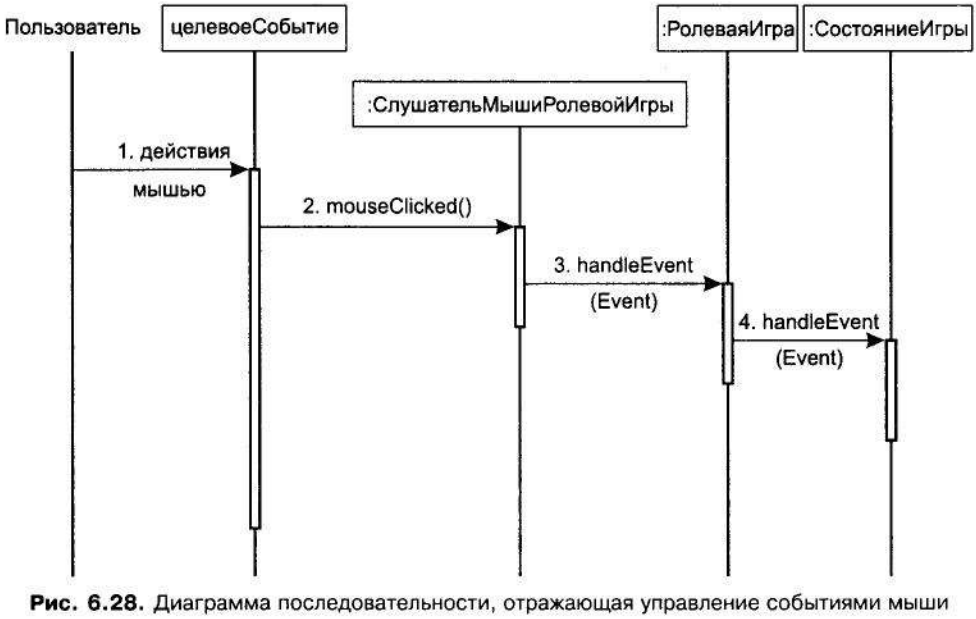


Рис. 6.28. Диаграмма последовательности, отражающая управление событиями мыши

[Примечание для студентов. Здесь можно поместить псевдокод методов, содержащихся в выделенных классах, а также описание вариантов использования. Поскольку в нашем случае методы и их подробное описание занимают лишь одну-две строки, нам достаточно подписей на рис. 6.28.]

6.1.2. Пакет Персонажи

Данный раздел содержит уточнения к разделу 3.1.2 SDD.

В пакете *Персонажи* имеется единственный класс: *ПерсонажИгры*.

6.1.2.1. Класс ПерсонажИгры

Метод `setName()` класса *ПерсонажИгры*.

Предусловия — нет; постусловия — нет; инварианты — нет. Псевдокод:

```
IF не задан параметр aName OR не задан параметр maxNumChrsInName()
    Установить имя по умолчанию и показать его в системном окне.
ELSE
    IF параметр string имеет слишком большую длину
        укоротить в соответствии с maxNumCharsInName()
    ELSE присвоить имя параметру
```

6.1.3. Пакет ИгроваяСреда

Этот пакет описан на рис. 5.36 в разделе 3.1 SDD.

6.1.4. Пакет Артефакты

[Отсутствует на данной итерации.]

[Окончание детального проектирования для пакетов каркаса.]

Пример 2. Ролевая видеоигра Встреча. Детальное проектирование (оставшиеся части SDD)

[Это продолжение документа SDD, оглавление которого вы найдете выше в разделе 6.7.1.]

6. Детальное проектирование ролевой видеоигры Встреча

Архитектура, отражающая связи между пакетами и классами предметной области, описываемыми в этом разделе, приводится на рис. 6.29.

6.1. Детальное проектирование модулей

[Примечание для студентов. В данном разделе приводятся все необходимые детали каждого из модулей, описанных в разделе 3.1 SDD.]

6.1.1. Пакет ИграВстреча

[Примечание для студентов. Здесь мы детализируем раздел 3.1.1 SDD, описывая классы пакета ИграВстреча. При этом рассматриваются только нетривиальные аспекты классов. Основным средством для описания являются диаграммы переходов состояний.]

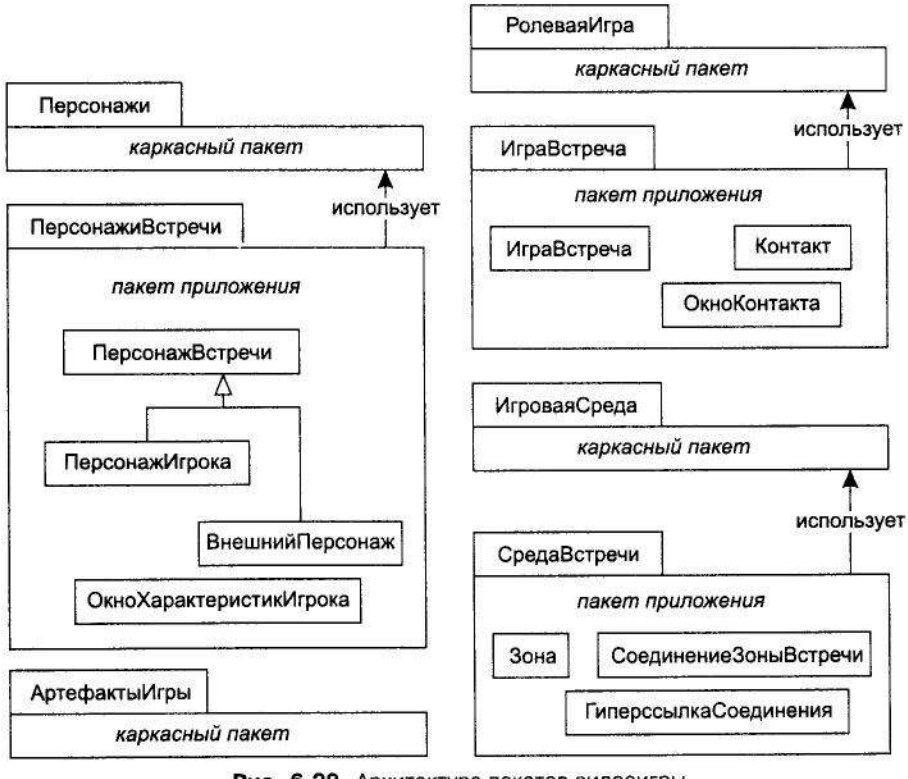


Рис. 6.29. Архитектура пакетов видеоигры

Диаграмма переходов состояний для класса *Встреча* показана на рис. 3.11. В соответствии с ней разработана объектная модель пакета *ИграВстреча* (рис. 6.30). В рамках из сплошных линий на рис. 6.30 представлены классы предметной области. Существует единственный экземпляр объекта класса *ИграВстреча*. Он проходит через те же состояния, что показаны в верхней части диаграммы переходов состояний. Состояние *Встреча* агрегирует объект *Контакт*, инкапсулирующий вступление персонажей игры в контакт. Класс *Контакт*, в свою очередь, агрегирует класс *ОкноКонтакта*. Последний чувствителен к событиям мыши, которые регистрирует объект *СлушательМышиРолевойИгры*. В процессе выполнения игры этот объект-слушатель ссылается на объект *ИграВстреча*, что позволяет объекту *ИграВстреча* обрабатывать события с учетом текущего состояния. Для этого используется образец проектирования State. Пакет *ИграВстреча* отвечает за перемещение внешнего персонажа. Эту задачу выполняют методы класса *ПеремещениеПерсонажа*, который является классом потоков.

Для реализации метода `handleEvent()` в соответствии с образцом проектирования State классам необходимо иметь возможность обращаться к другим пакетам. Эта цель достигается благодаря использованию facade-объектов *РолиВстречи* и *СредаВстречи*.

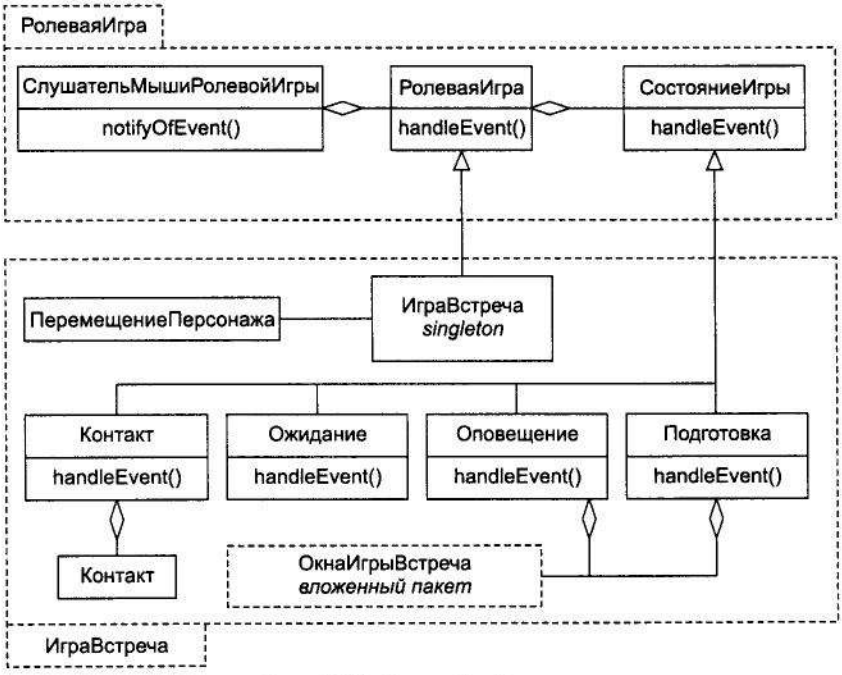


Рис. 6.30. Пакет ИграВстреча

6.1.1.1. Пакет ОкнаИгрыВстреча, вложенный в пакет ИграВстреча

Для управления видами экрана, относящимися к тому или иному состоянию, существует отдельный вложенный пакет *ОкнаИгрыВстреча* (рис. 6.31). `QualListDispl` — это поле со списком, содержащее характеристики персонажа *Встречи*. `QualValueDispl` — это текстовое поле, доступное только для чтения, в котором выводится значение характеристики. Для ввода значения характеристики предназначено редактируемое поле `SetQualValueDispl`. *ЭлементОкнаВстречи* является абстракцией, в которой собраны свойства и методы отображаемых на экране элементов *Встречи*.

Класс *ОкноКонтакта* разрабатывается для отображения текущего значения выделенной характеристики. Класс *ОкноХарактеристикИгрока* позволит игроку задавать значение любой характеристики. *ОкноВстречи* обобщает их свойства и представляет собой базовый класс (см. образец проектирования Mediator).

Более детально в данном документе эти классы не рассматриваются.

6.1.1.2. Диаграммы последовательности для обработки событий

6.1.1.2.1. Событие: Игрок закрывает окно контакта

Последовательность действий, которой сопровождается закрытие игроком окна со сведениями о контакте, показана на рис. 6.32.

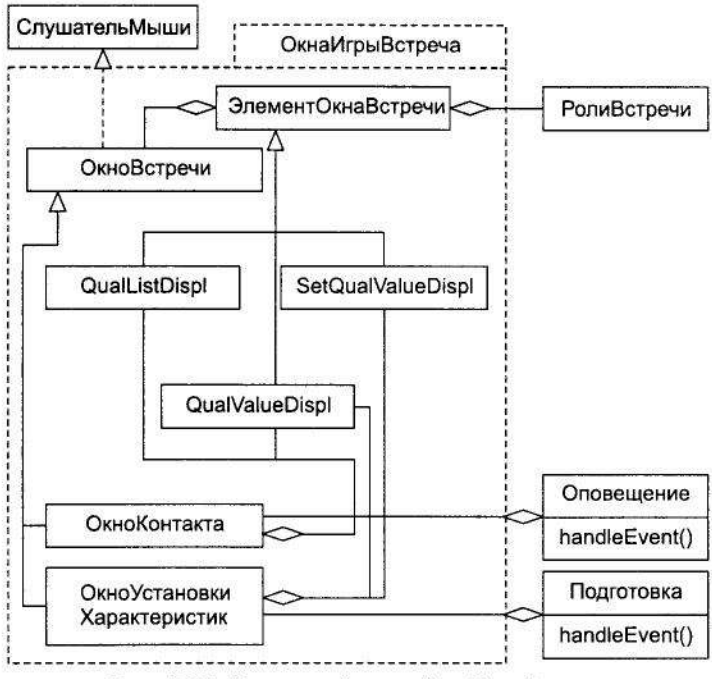


Рис. 6.31. Вложенный пакет ОкнаИгрыВстреча

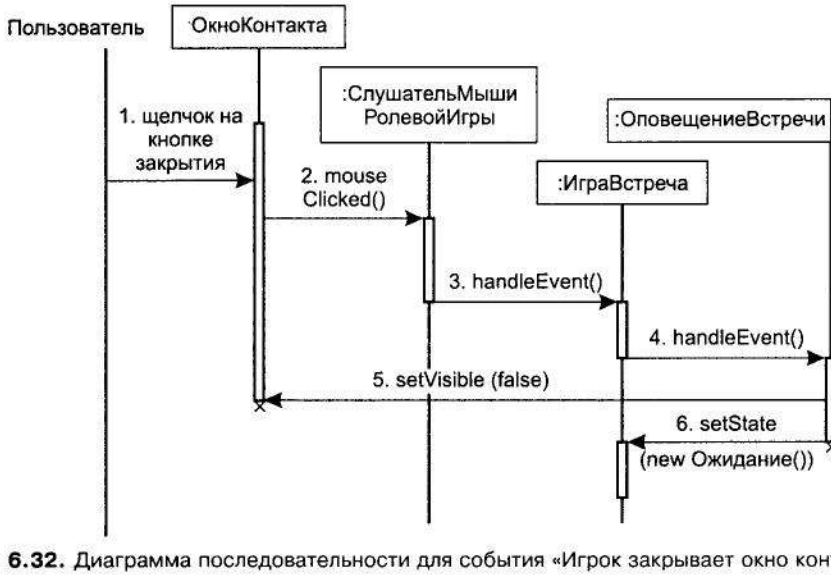


Рис. 6.32. Диаграмма последовательности для события «Игрок закрывает окно контакта»

6.1.1.2.2. Событие: Игрок завершает установку характеристик

Последовательность действий, которой сопровождается завершение установки характеристик игроком, показана на рис. 6.33.

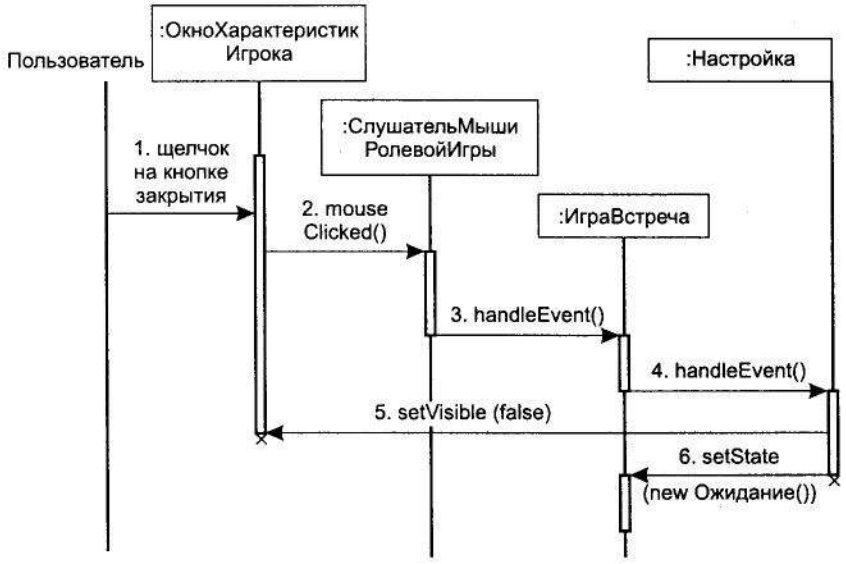


Рис. 6.33. Диаграмма последовательности для события «Игрок завершает установку характеристик»

6.1.1.2.3. Событие: Игрок перемещается в соседнюю зону

Диаграмма последовательности для того случая, когда игрок переходит в соседнюю зону, щелкнув на соответствующей гиперссылке, показана на рис. 6.34.

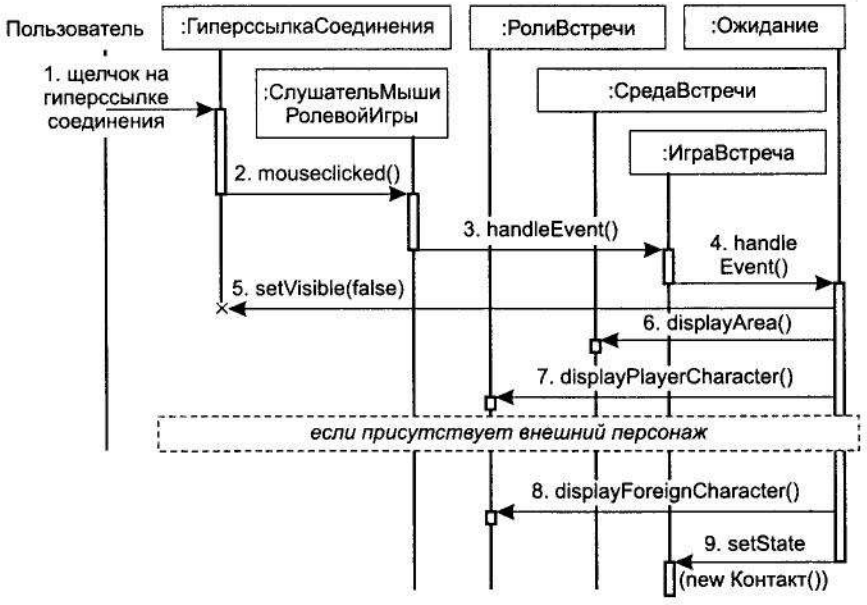


Рис. 6.34. Диаграмма последовательности для события «Игрок перемещается в соседнюю зону»

6.1.1.2.4. Диаграммы последовательности для других событий

Остальные события обрабатываются способом, аналогичным описанному в разделах 6.1.1.1–6.1.1.3.

6.1.1.ИВ. Класс ИграВстреча

Это одиночный facade-класс пакета *ИграВстреча*.

Наследование

Этот класс наследуется от каркасного пакета *РолеваяИгра*.

Атрибуты класса ИграВстреча

```
EncounterGame encounterGameS
```

Это одиночный объект *ИграВстреча*.

Конструкторы класса ИграВстреча

```
private EncounterGame()
```

Предусловия: нет.

Постусловия: создает экземпляр *ИграВстреча*.

Методы класса ИграВстреча

```
public static EncounterGame getTheEncounterGame();
```

Предусловия: нет.

Постусловия: значение `encounterGameS` отлично от нуля.

Возвращаемое значение: `encounterGameS`.

6.1.1.КО. Класс Контакт

Этот класс инкапсулирует контакты между персонажем игрока и внешним персонажем согласно требованию SRS 3.2.КО.

Методы

```
public void engage(); // согласно требованию 3.2.КО.3.1
```

Предусловия: персонаж игрока и внешний персонаж находятся в одной зоне.

Постусловия: значения характеристик персонажей соответствуют требованию SRS 3.2.КО.3.1; персонаж игрока и внешний персонаж находятся в случайно выбранных, но различных зонах.

6.1.1.ПП. Класс ПеремещениеПерсонажа

[Примечание для студентов. В этом документе, как и в SRS, мы используем буквенные обозначения для именования подразделов. Это позволяет упростить поиск и добавление классов. Еще один плюс такой «нумерации» в том, что мы можем прослеживать здесь требования SRS.]

Данный класс управляет перемещением внешнего персонажа.

Наследование

Данный класс наследуется от `java.lang.Thread`.

Методы

```
public static EncounterGame run();
```

Изначально помещает внешний персонаж в зону *подвал*. Затем перемещает внешний персонаж из одной зоны в другую согласно соединениям между зонами. Соседние зоны для перехода выбирает случайным образом через случайные временные интервалы, в среднем равные 2 секундам.

6.1.1.ЯЯ. Классы состояния Контакт, Ожидание, Подготовка и Отчет

[Сокращением «ЯЯ» мы обозначили группу классов, которые мы не будем описывать по отдельности.]

Наследование

Эти классы наследуются от класса *СостояниеИгры* из каркасного пакета.

В каждом из этих классов согласно диаграммам последовательности из раздела 6.1.1.1 реализуется метод `handleEvent()`.

6.1.2. Пакет ПерсонажиВстречи

[Этим разделом уточняется раздел 3.1.2 SDD.]

Структура пакета *ПерсонажиВстречи* показана на рис. 6.35. Он реализуется с применением образца проектирования Facade, при этом в качестве facade-объекта используется объект *РолиВстречи*.

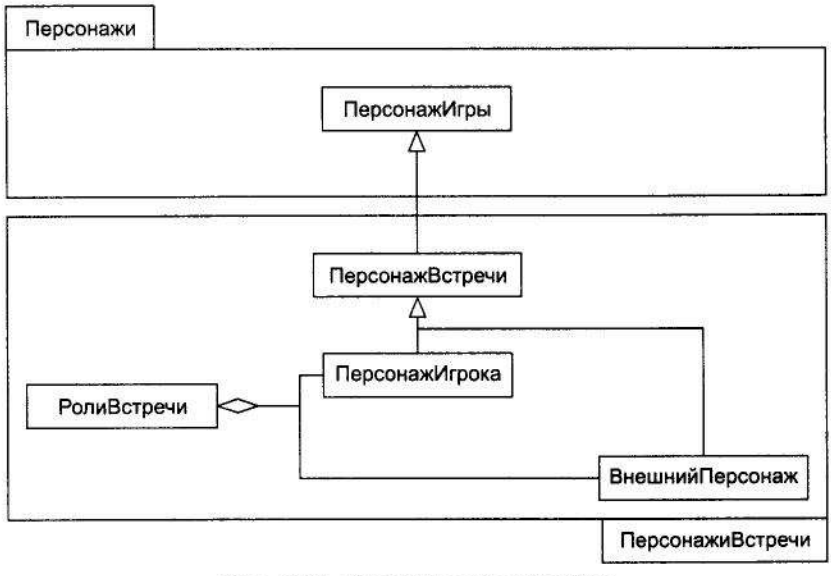


Рис. 6.35. Пакет ПерсонажиВстречи

6.1.2.ВП. Класс ВнешнийПерсонаж

Этот класс аналогичен описываемому далее классу *ПерсонажИгрока*. Он разрабатывается согласно требованиям SRS 3.2.ВП.

6.1.2.ПВ. Класс ПерсонажВстречи

В этом классе инкапсулируются требования к персонажам *Встречи*, не заданные в пакете *ПерсонажиИгры*, согласно требованию SRS 3.2.ПВ.

Инвариант класса: значения qualValueI неотрицательны (описание qualValueI см. далее в перечне атрибутов класса).

Наследование

Этот класс наследуется от каркасного пакета *ПерсонажиИгры*.

Атрибуты класса ПерсонажВстречи

Согласно требованиям SRS 3.2.ПВ.1.

```
private static final String[] qualityTypes
```

Представляет характеристики, которыми обладают персонажи *Встречи*. Это *сосредоточенность*, *выносливость*, *ум*, *терпение* и *сила*.

```
private float[] qualValueI
```

Массив значений характеристик.

Конструкторы класса ПерсонажВстречи

Согласно требованиям SRS 3.2.ПВ.3.

Null-конструктор.

Постусловия: значения характеристик представляют собой равные доли числа 100.

```
protected EncounterCharacter( String nameP )
```

Постусловия:

- значения характеристик представляют собой равные доли числа 100;
- имя персонажа равно содержимому NameP.

Методы класса ПерсонажВстречи

```
public synchronized void adjustQuality( String qualityP, float qualityValueP )
```

Этот метод удовлетворяет требованию SRS 3.2.ПВ.3.2.

Инварианты: нет.

Предусловия:

qualityP принадлежит qualityTypesS[]

AND qualityValueP >= 0

AND qualityValueP <= сумма значений характеристик

Постусловия:

qualityP имеет значение qualityValueP

AND

Значения остальных характеристик остаются в том же соотношении, что и раньше, за исключением значений меньших 0,5 — они приравниваются к нулю.

Далее приводится псевдокод метода adjustQuality().

```
Присвоить характеристике требуемое значение.
IF   Настраивается единственная ненулевая характеристика,
     разделить введенное число поровну между всеми остальными
     характеристиками.
ELSE Изменить остальные характеристики так, чтобы сохранилось соотношение
     характеристик.
```

```
Обнулить характеристики, значения которых теперь меньше 0,5.
public float getQualityValue( String qualityP )
```

Предусловия: qualityP — строка, содержащая корректное имя характеристики.
Возвращаемое значение: значение qualityP.

```
public float getTolerance( )
```

Возвращаемое значение: пороговое значение ненулевых характеристик.

```
public int maxNumCharsInName( )
```

Возвращаемое значение: максимальное число символов, разрешенное в имени персонажа *Встречи*.

```
public float sumOfQualities( )
```

Возвращаемое значение: сумма значений характеристик.
Этот метод удовлетворяет требованию SRS 3.2.ПВ3.2.

```
public void showCharacter( Component compP, Graphics drawP, Point posP,
    int heightPixP, boolean faceLeftP )
```

Отображает персонаж в компоненте пользовательского интерфейса compP с центром в точке posP, высотой heightPixP и, если значение faceLeftP истинно, смотрящим влево.
Этот метод удовлетворяет требованиям 3.2.ПИ.1 и 3.2.ХИ.1.

```
private void setQuality( String qualityP, float valueP )
```

Предусловия: qualityP — строка, содержащая корректное имя характеристики.
Присваивает указанной характеристике значение, заданное параметром valueP, если он больше или равен 0,5. В противном случае характеристика обнуляется.
Данный метод удовлетворяет требованию 3.2.ПВ.3.2 (нижняя граница ненулевых значений характеристик).

6.1.2. ПИ. Класс ПерсонажИгрока

Данный класс разрабатывается согласно требованиям SRS 3.2.ПИ.

Наследование

Данный класс наследуется от класса *ПерсонажВстречи*.

Атрибуты

```
private static final PlayerCharacter playerCharacterS;
```

Это одиночный объект, представляющий персонаж игрока.

Методы

```
public static final PlayerCharacter getPlayerCharacter();
```

Этот метод возвращает playerCharacterS.

6.1.2.РВ. Класс РолиВстречи

Метод, дающий представление об этом одиночном интерфейсном классе, приведен в разделе 5 данного документа.

6.1.3. Пакет СредаВстречи

Классы этого пакета описывают окружение, в котором проходит игра (рис. 6.36).

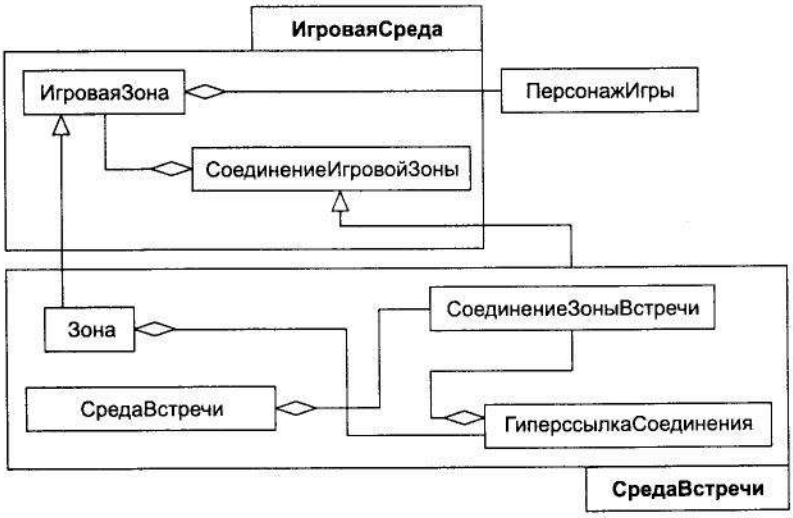


Рис. 6.36. Пакет СредаВстречи

6.1.3.ГС. Класс ГиперссылкаСоединения

Этот класс инкапсулирует способы перехода в соседние зоны. Он разрабатывается согласно требованию SRS 3.2.ГС. В этом классе реализуется интерфейс *СлушательМыши*.

Атрибуты

```
private Connection connectionI; // соответствующее соединение зон
```

Методы

Единственный метод, достойный упоминания, — `mouseClick()`.

6.1.3.ЗО. Класс Зона

Этот класс инкапсулирует области, в которых существуют персонажи, и разрабатывается в соответствии с требованием SRS 3.2.ЗО.

Наследование

Этот класс наследуется от класса *ИгроваяЗона*.

Атрибуты

```
private String nameI; // согласно требованию 3.2.30.1.1
private Image imageI; // согласно требованию 3.2.30.1.2
private String[] qualitiesI; // согласно требованию 3.2.30.1.3
private Vector connectionHyperlinksI;
```

Методы

```
public void display()
```

Выводит изображение объекта-зоны на экран.

```
public static Area getArea( String areaNameP )
```

Возвращает объект-зону, соответствующую **areaNameP**. Удовлетворяет требованию 3.2.ЗО.2.

```
public static AreaConnection( String areaConnectionNameP )
```

Возвращает объект **AreaConnection**, соответствующий **areaConnectionNameP**. Удовлетворяет требованию 3.2.ЗО.2.

6.1.3.СВ. Класс СоединениеЗоныВстречи

Этот класс инкапсулирует способы переходов в соседние зоны. Он наследуется от класса *СоединениеЗоны* и разрабатывается согласно требованию SRS 3.2.СВ.

Наследование

Этот класс наследуется от класса *СоединениеИгровойЗоны* каркасного пакета.

Атрибуты

```
private Area firstAreaI;  // согласно требованию 3.2.CB.1.1
private Area secondAreaI; // согласно требованию 3.2.CB.1.1
```

Методы

Предоставляют доступ к указанным выше атрибутам.

6.1.3.СР. Класс СредаВстречи

Представляет собой facade-класс для пакета *СредаВстречи*.

Атрибуты

```
private EncounterEnvironment encounterEnvironmentS;
// одиночный facade-объект
//[Имя зоны][Имя соединения зон][«North»|"South"|"East"|"West"]:
private String[3] layoutS;
```

Методы

```
public static EncounterEnvironment getEncounterEnvironment()
```

Возвращаемое значение: **encounterEnvironmentS**.

```
public static String[3] getLayout()
```

Возвращаемое значение: **layoutS**.

6.2. Детальное проектирование данных

Никаких структур данных, кроме упомянутых при описании классов в разделе 6.1, не имеется.

Глава 7
Реализация модулей

В чем тайный смысл такой горячей спешки,
Что стала ночь сотрудницею дня?
Шекспир. Гамлет

Написание программ на скорую руку дает быстрый, но сомнительный результат; дисциплинированный подход, напротив, обеспечивает более высокое качество с меньшими временными затратами.

В этой главе не предпринимается попытка охватить все аспекты программирования в духе таких книг, как [44]. Вместо этого мы обсудим принципы программирования и дадим рекомендации, которые относятся в большей степени к компаниям, разрабатывающим программные приложения. Положение этой главы в контексте разработки программного обеспечения показано на рис. 7.1.

- Разделы 7.1–7.7
- Упражнения

Рис. 7.1. План разработки программ: темы главы 7

- Примеры
  1) Обновление Плана контроля качества программного обеспечения (SQAP)
  2) Обновление Плана управления конфигурациями программного обеспечения (SCMP)
  3) Индивидуальная программная документация (PSD), часть 1
  4) Пример исходного кода для класса EncounterCharacter (ПерсонажВстречи)

В этой главе рассматриваются следующие вопросы:

- Выбор стандартов.
- Выбор стиля кодирования.
- Кодирование с обоснованием корректности.
- Определение требований качества.

7.1. Введение в реализацию

7.1.1. Определение «реализации модулей»

Термин «реализация» относится к программированию. Термин «модуль» относится к самым мелким частям реализации, которые можно поддерживать отдельно: это может быть отдельный метод или класс. В примерах мы будем считать методы модулями самого низкого уровня.

7.1.2. Цели реализации

Целью реализации является удовлетворение требований способом, определенным в детальном проектировании. Детального проекта должно быть достаточно, поскольку это документ, на котором основывается программирование. Однако часто программист исследует параллельно все предшествующие документы (архитектуру, D-требования и C-требования), что помогает сгладить несогласованность этих документов.

7.1.3. Типичная схема процесса реализации модулей

Типичный процесс написания кода программы показан на рис. 7.2.

1. Стандарты кодирования на рис. 7.2 определены так, чтобы исходный код имел стандартный вид (раздел 7.3).
2. Архитектура определяет каркас и пакеты прикладных программ. Каждый класс каждого пакета затем реализуется через кодирование методов, определенных требованиями в детальном проектировании. Пакеты структуры необходимо иметь до начала разработки программы.

3. Каждый класс инспектируется, как только он завершен.
4. Затем каждый класс тестируется, как описано в главе 8.
5. Пакет или классы затем выпускаются для интеграции в разрабатываемое приложение.

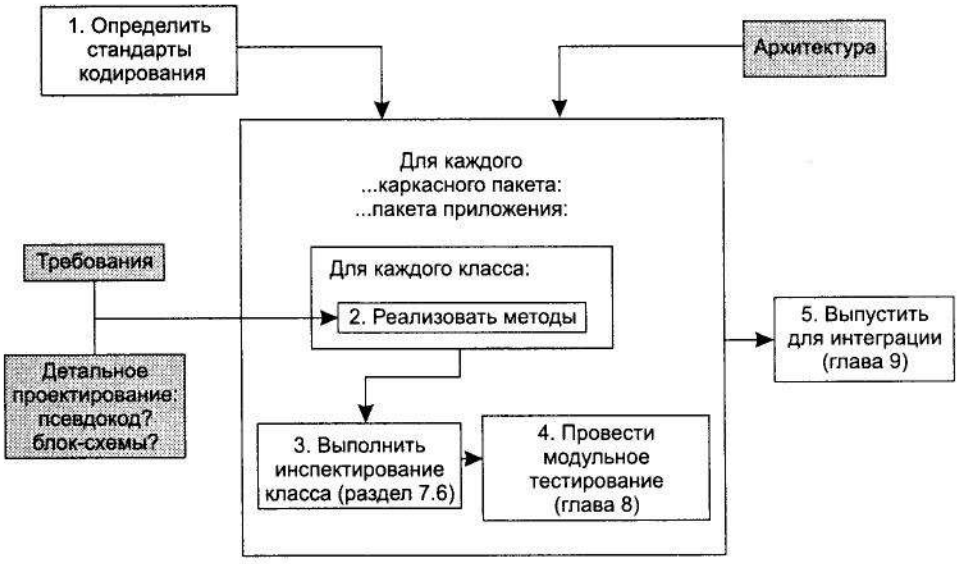


Рис. 7.2. План реализации модуля

ОДИН ИЗ СПОСОБОВ ПОДГОТОВИТЬСЯ К РЕАЛИЗАЦИИ

1. Подтвердите детальное проектирование, которое вы должны реализовать.
   - Кодируйте только записанные этапы проектирования (части SDD).
2. Подготовьтесь измерить затраченное время, классифицируя по следующим пунктам:
   - остаточное детальное проектирование; обзор детального проектирования; кодирование; обзор кодирования; компилирование и исправление синтаксических дефектов; тестирование модулей (глава 8) и исправление дефектов, найденных при тестировании.
3. Подготовьтесь записать дефекты, упорядочив их по двум характеристикам:
   - серьезность: высокая (требование не выполнено), низкая или ни то, ни другое;
   - тип: ошибка, присваивание имен, окружение, система, данные, другой.
4. Стремитесь понимать, какие стандарты требуются:
   - для кодирования;
   - для персональной документации, которую вы должны хранить.
   - См. учебный пример в конце главы.
5. Оцените объем работ и время на основе ваших предыдущих данных.
6. Планируйте работу сегментами порядка 100 строк кода.

Некоторые из наиболее сложных методов будут приведены в псевдокоде или блок-схемах. Псевдокод из документа детальных требований может стать ком-

ментарием к коду. План тестирования разрабатывается для каждого модуля, как описано в следующей главе. Каждый из модулей, показанных на диаграмме, тестируется. Когда класс реализован, можно использовать инструменты обратного проектирования (например, Javadoc) для повторной генерации аспектов детального проектирования (рис. 7.3).

Рис. 7.3. Использование обратного проектирования в детальном проектировании

Обратное проектирование можно посоветовать после начальной реализации, поскольку исходный код часто становится более современным, чем нижний уровень детального проектирования. Это также может оказаться полезным для унаследованного кода, который плохо документирован. Вообще говоря, обратное проектирование нужно использовать только когда оно действительно необходимо. Автор наблюдал много ситуаций, когда обратное проектирование использовалось для скрытия недостатков проекта. Другими словами, инженеры преждевременно «ныряют» в написание кода, а затем подтверждают его разработкой обратного «проекта», полученного из кода. Читателей таких «проектов» редко удается обмануть.

Обратное проектирование также обсуждается в главах 6 и 10.

7.1.4. Реализация в USDP

Вспомните, что USDP дает названия группам итераций. Большинство итераций слабо влияют на реализацию, однако итерации конструирования затрагивают основную часть реализации [64] (рис. 7.4).

USDP рассматривает реализацию как еще одну модель (наряду с моделями вариантов использования, тестовой моделью и т. д.). Хотя составные части проекта и должны отображаться как можно более точно на соответствующие части физической файловой системы, простое отображение может быть непрактично. Например, несколько классов можно отобразить на один файл, и артефакты реализации могут содержать ознакомительные файлы `README`, исходные файлы, объектные файлы и сжатые версии, такие как JAR-файлы. *Модель реализации* показывает,

как организованы физические артефакты реализации и как на них отображаются элементы проекта (рис. 7.5).

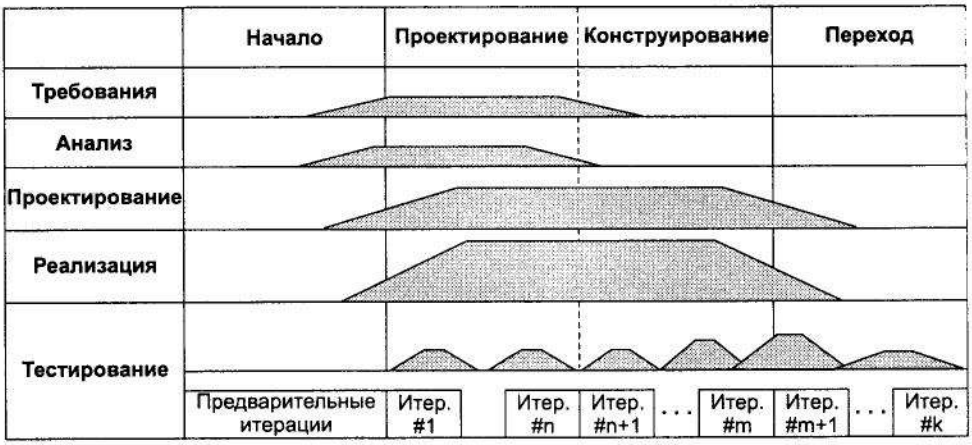


Рис. 7.4. Реализация в USDP

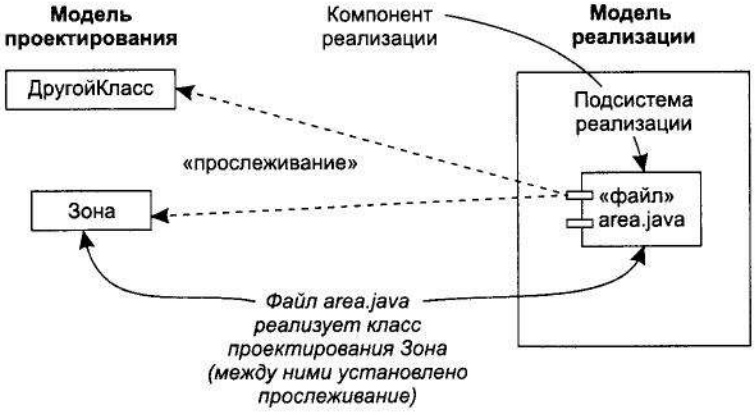


Рис. 7.5. Компоненты модели реализации USDP (1)

Модель показывает, что классы *Зона* и *ДругойКласс* реализованы в одном файле — Area.java (*ДругойКласс* может быть внутренним по отношению к классу *Зона*).

Модель реализации состоит из вложенных подсистем. Они состоят из таких компонентов, как файлы и интерфейсы реализации (рис. 7.6). Исходный код для классов *Зона* и *ДругойКласс* (рис. 7.6) находится в файле Area.java. Интерфейс *Зоны* (набор функций типа public) в проекте соответствует реально существующей функциональности, реализованной в Area.java. Модель реализации также предоставляет нотацию для отображения связи между файлами, например «реализует», «описывает» или «сжимает», посредством кавычек (« »). (Организация может сама определять эти связи.) Например, на рис. 7.6 указано, что файл README, исходный код *Зоны* и его объектный код находятся в одной коллекции и что файл impl.jar — это их jar-версия (сжатый Java-архив).

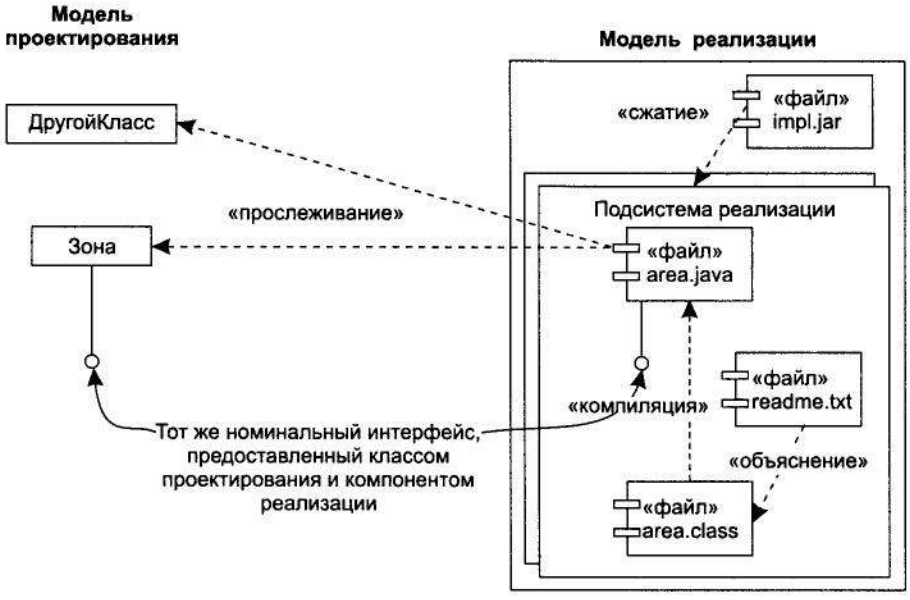


Рис. 7.6. Компоненты модели реализации USDP (2)

Модель реализации описывает интерфейс уровней классов и языка, отвечающий за реализацию системного интерфейса. Пример для видеоигры *Встреча* с использованием одного интерфейсного класса показан на рис. 7.7. С помощью этой нотации можно показать несколько классов, отвечающих за части интерфейса подсистем. Их можно объединить в интерфейсный пакет, отвечающий за интерфейс, и т. д.

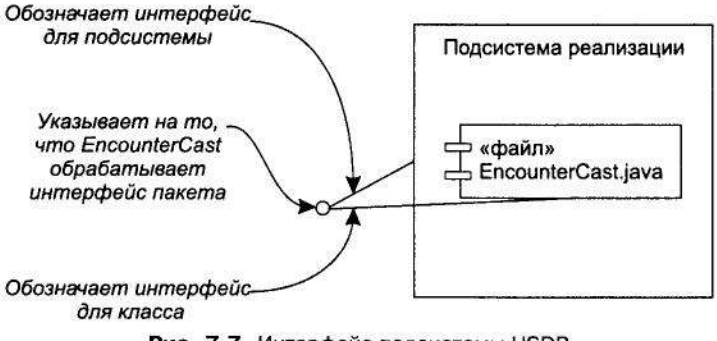


Рис. 7.7. Интерфейс подсистемы USDP

7.1.5. Языки программирования

Существует огромное количество языков программирования, от специализированных (например, языки тестирования аппаратуры) до универсальных языков высокого уровня, таких как C++, Java и COBOL. Развитие Интернета затрагивает

широкий спектр языков программирования. Принципы, описанные в этой книге, можно применять к любому из этих языков, однако объектно-ориентированные языки твердо придерживаются наиболее точного использования этих принципов. Предполагаемый язык реализации влияет на проект программного приложения. Например, при использовании такого языка, как C, может оказаться предпочтительнее разрабатывать проект более нисходящим, функциональным способом, несмотря на то что объектно-ориентированные проекты тоже могут в некоторой степени реализовываться на C. Некоторые языки, такие как Javascript и ранее Visual Basic, основываются на концепции объектов, что дает программистам возможность использовать инкапсуляцию и агрегирование, но без наследования.

7.2. Программирование и стиль

Популярное представление о программировании как о процессе подготовки текста программы для компиляции на самом деле отражает лишь незначительную часть общей картины. Действительной целью написания программы является создание *правильного* кода, то есть такого кода, который полностью соответствует сформулированным для нее требованиям. Компиляторы же только проверяют корректность синтаксиса и генерируют объектный код. За правильность программы отвечает человек, и поэтому настоящий профессионал должен убедиться в правильности своей программы до того, как ее компилировать. Конечно, можно и сначала компилировать, а затем убеждаться в правильности, но это так же неэффективно, как сначала делать синтаксическую правку письма, а потом проверять, насколько правильно в нем изложены мысли. Заметим также, что программисты обычно упорно не хотят анализировать свою программу после того, как она успешно откомпилирована.

Завершает этот раздел рекомендуемая последовательность шагов для написания программ, разработанная на основе материалов этой главы. Здесь выделены типичные советы по стилю программирования. Существует много хороших книг, содержащих советы по программированию, однако в этой главе мы не ставим перед собой цели повторить их все.

ОДИН ИЗ СПОСОБОВ РЕАЛИЗАЦИИ КОДА (1)

1. Спланируйте структуру вашего кода.

   Восполните пробелы в детальном проектировании, если таковые имеются.

   - Отметьте предусловия и постусловия;
   - отметьте затраченное время.

2. Проинспектируйте проект и (или) структуру.
   - Отметьте затраченное время, тип, серьезность и источник дефектов.
3. Наберите код:
   - пока не компилируйте;
   - попробуйте использовать методы, перечисленные ниже;
   - примените необходимые стандарты;
   - кодируйте таким способом, который будет легче всего проверить (используйте формальные методы).

ОДИН ИЗ СПОСОБОВ РЕАЛИЗАЦИИ КОДА (2)

1. Проинспектируйте код, но пока не компилируйте:
   - убедитесь, что ваш код выполняет требуемую работу (компилятор никогда не сделает это за вас: он просто проверит синтаксис!);
   - отметьте затраченное время, тип, серьезность и источник дефектов;
   - просмотрите контрольный список инспектирования (начало раздела 7.6) для часто используемых деталей конструирования методов и классов.
2. Откомпилируйте код:
   - исправьте синтаксические ошибки;
   - отметьте затраченное время, тип, серьезность и источник дефектов, число строк кода.
3. Протестируйте код.

   Примените методы модульного тестирования из главы 7.

7.2.1. Общие принципы надежной реализации

Рекомендуется учитывать два общих принципа в программировании.

1. Сначала постарайтесь повторно использовать имеющийся код.
2. Принцип *«осуществления намерений»*.

   Если вы планируете использовать ваш код только в каких-то частных случаях, пишите его так, чтобы этот код было невозможно использовать никак иначе.

Поясним эти принципы подробнее.

1. Мы сделали акцент на необходимости разрабатывать проекты наших собственных программ так, чтобы разрешить повторное использование построенных нами компонентов. Аналогично, мы будем рассматривать возможность повторного использования существующего надежного кода, прежде чем начать писать новый. Сравните, например, использование компонента Java Swing GUI или существующего Java Beans. На самом деле поиск компонентов для повторного использования в Интернете обычно дает хорошие результаты.
2. Если вы размышляете над тем, как ваш код будет использоваться другими частями программы, постарайтесь претворить это ваше намерение в жизнь. Автор называет это принципом «осуществления намерений». Часто из пользовательского интерфейса очевидно, где пользователю не разрешено вводить запрещенные данные. Однако мы относим это к внутренней обработке.

Принцип «осуществления намерений» аналогичен конструированию обочин и разделительных полос на дорогах для направления движения только по тем путям, по которым это было запланировано инженерами движения, и никаким другим. Такое принуждение делает дороги более безопасными и широко используется в каждой инженерной отрасли. Далее приведены примеры принципа «осуществления намерений» в разработке программ.

- Используйте спецификаторы, такие как `final`, `const` в C++ и `abstract`, для осуществления соответствующих намерений. Классы `final` не могут быть унаследованы; `final`-методы нельзя переписать в унаследованных классах.

значения переменных `final` нельзя изменить. Если это приводит к ошибкам во время компиляции, это означает, что вы не до конца осознаете вашу собственную программу и никакого вреда еще не причинили. Особенно мы стараемся избежать ошибок во время запуска.

- Сделайте константы, переменные и классы настолько локальными, насколько это возможно в пределах разумного. Например, определяйте счетчики циклов в цикле: не давайте им более широкой области использования.
- Используйте образец проектирования одноэлементных множеств Singleton, если предполагается только один объект класса (см. главу 6).
- В общем случае делайте члены класса недоступными, если нет отдельного требования обеспечить прямой доступ к ним.
- Делайте атрибуты скрытыми (`private`). Доступ к ним реализуйте по необходимости через функции общего доступа. (В Java установка типа `protected` для атрибутов дает объектам или подклассам доступ к членам их базового класса, что часто нежелательно.)
- Делайте методы скрытыми, если они должны использоваться только методами того же класса.
- Добавьте в документацию примеры. Программисты обычно колеблются на счет этого, однако примеры могут значительно помочь читателю. В нашем учебном проекте пример приведен в комментариях к методу `adjustQuality()` класса `EncounterGame` (ИграВстреча).
- Перечисляйте методы в алфавитном порядке. Не пытайтесь найти порядок их вызова. Некоторые программисты любят группировать методы `private`, `protected` и `public`, а затем разбивают их на подгруппы статических и нестатических методов (в нашем примере эта практика не используется).

7.2.2. Указатели и ссылки

Далее перечислены рекомендации, сформулированные в [47].

- Избегайте использования переменных-указателей в C++ — вместо этого используйте ссылки.
- Никогда не возвращайте указатель на новую ссылку на «кучу» в C++. Иначе вы будете ссылаться на неиспользуемую память, и к обязанностям программиста добавится восстановление пространства памяти.
- Производите сбор мусора (C++), используя оператор `delete()` для всех ненужных объектов. Неудача при восстановлении памяти (утечка памяти) является основной проблемой в программах на C++. Некоторые организации, занимающиеся разработкой на C++, позволяют программистам выделять память только посредством специальных служебных функций, чтобы улучшить контроль над процессом выделения памяти. Другими словами, они стараются избегать `new c()`: программисты должны использовать такие утилиты, как `F::getNewCObject()` для некоторого класса `F`.

- Существуют также коммерческие инструменты, пытающиеся определить потенциально возможные утечки памяти. Сбор мусора в Java производится автоматически, так что утечка памяти такого масштаба, как в C++, невозможна. Однако Java-программы могут накапливать такие ресурсы, как файлы и сокеты, и программист должен вести учет ресурсов, которые его программа использует во время выполнения.

7.2.3. Функции

- Избегайте запросов типов, например `if ( myObject instanceOf MyClass )`, за исключением случаев, когда это действительно необходимо. Вместо этого используйте виртуальные функции. (См., например, образец проектирования State в главе 5.)
- В C++ избегайте дружественных функций (`friend`) за исключением случаев, когда их достоинства явно преобладают над недостатками.
- Обратите особое внимание на перегрузку операций, поскольку когда вашу программу будут читать другие люди, они могут неправильно понять смысл ваших операций. В связи с этим Java не разрешает перегрузку.

7.2.4. Исключения

Перехватывайте только те исключения, для которых вам известен способ обработки. Метод, обрабатывающий исключение, выглядит так:

```
ReturnType myMethod( … ) { …
    try { … // вызвать метод, генерирующий ExceptionX }
    catch( ExceptionX e ) {… // обработать его }
    …}
```

Метод, не обрабатывающий исключение, выглядит так:

```
ReturnType myMethod(…) throws ExceptionX{…}
```

- Если существующий метод не может обработать исключение, должен быть обработчик во внешнем коде, который может это осуществить.
- Если вы можете обработать часть исключения, обработайте ее, а затем повторно сгенерируйте исключение для его обработки во внешнем коде.
- Делайте разумные предположения относительно возможности вызывающих операторов обработать ваше исключение или найдите альтернативный вариант разработки, поскольку необработанные исключения приводят к аварийным ситуациям в программе.
- Будьте аккуратны и не используйте исключения в ситуациях, которые могут быть предметом тестирования. Например, если предусловие говорит, что метод никогда не должен вызываться с нулевым параметром, это необходимо проверить *тестированием*. Генерация исключения в случае ненулевого параметра не должна быть стандартным способом обработки такой проблемы.

- «Если вам нужно выбрать между генерацией исключения и продолжением вычислений, продолжайте вычисления, если можете» [47]. Здесь имеется в виду, что программу имеет смысл закрыть только в тех случаях, когда вы тщательно продумали последствия.

  Например, в нашем учебном проекте видеоигры программа продолжает работать с именем, присвоенным по умолчанию, если задан неразрешенный строковый параметр (например, `null`). Это лучше, чем закрывать игру только из-за ввода неразрешенного имени. Однако банковская операция с неразрешенной суммой не должна обрабатываться.

В [47] указаны несколько причин, которые могут привести к неправильной работе конструкторов:

- первой причиной ошибки конструктора может быть неправильный аргумент. В [47] рекомендуется установить объект в некоторое состояние по умолчанию и только затем передавать управление обработчику ошибок;
- вторая причина — недоступность необходимых ресурсов. Эту ситуацию лучше всего обработать через генерацию исключения, поскольку мы ничего не можем с этим сделать;
- программисты (особенно на C++) должны быть особенно внимательны при работе с памятью, которая будет оставлена после генерации исключения в ответ на ошибку конструктора.

7.2.5. Обработка ошибок

Разработчики постоянно сталкиваются с проблемой обработки некорректных данных. Примером некорректных данных может быть номер счета, не соответствующий существующему банковскому счету. Несмотря на то что мы стремимся делать реализации как можно более простыми, окружающий нас мир не так уж прост. Довольно большая часть процесса программирования связана с обработкой ошибок. Здесь очень важен дисциплинированный подход: выберите подход, сформулируйте его и убедитесь, что все остальные члены команды понимают его и будут его придерживаться. Один из способов обработки ошибок показан ниже. Он будет объяснен далее.

ОДИН ИЗ СПОСОБОВ РЕАЛИЗАЦИИ ОБРАБОТКИ ОШИБОК

1. Следуйте заранее определенному процессу разработки; выполняйте инспектирование.
2. Вводя классы, проверяйте корректность инкапсулируемых значений параметров:
   - private-конструктор, factory-функции для создания объектов;
   - при компиляции можно отловить достаточно много ошибок.
3. Если обработка ошибок определена в требованиях, реализуйте ее согласно требованиям. Используйте исключения при передаче управления во время обработки ошибок.
4. Для программ, которые никогда не должны давать отказ, старайтесь предвидеть все возможные дефекты реализации (например, используйте значения по умолчанию).
5. Альтернативой может быть следование согласованной политике проверки параметров. В основном полагайтесь на качественное проектирование и разработку.

Нашей основной целью является *предотвращение появления* ошибок, а не их исправление. Использование четко определенного процесса, инспектирование этапов и т. д. являются жизненно важной первой линией обороны. Некоторые образцы проектирования также могут помочь предотвратить ошибки. Например, если метод `evaluate()` принимает в качестве параметров только строки «car», «truck» и «bus», имеет смысл отказаться от использования параметра `String`, поскольку оно может привести к появлению некорректных параметров. Правильнее будет определить класс, например `SpecializedVehicle`, со скрытым конструктором и функциями образца проектирования Factory:

```
SpecializedVehicle createCar()
SpecializedVehicle createTruck()
SpecializedVehicle createBus()
```

Тогда наш метод будет принимать параметры только этого типа. Другими словами, вместо конструктора:

```
evaluate( String vehicleP ) … //проблема с некорректными String-параметрами
```

необходимо использовать:

```
evaluate( SpecializedVehicle vehicleP ) … //значение параметра не может быть
некорректным
```

Когда возможные значения параметра ограничены, но не определены, все еще можно использовать отдельный класс. Например, если возраст человека будет представлен целым числом от 0 до 105, то метод

```
getYearOfBirth( int ageP )
```

может получить неправильный параметр. Более того, обработку ошибок придется повторить для всех методов, получающих в качестве параметра возраст. Ниже описаны несколько вариантов решения этой проблемы. Их недостаток заключается в быстром увеличении количества дополнительных классов и некоторой странности вызовов, например, когда мы используем вызов:

```
… getYearOfBirth( getAge( n ) ) …
```

вместо такого простого вызова:

```
… getYearOfBirth( n ) …
```

Второй линией защиты при работе с возможно некорректными данными является взаимодействие с источником данных до тех пор, пока они не будут приведены к разрешенному виду, которое должно предшествовать дальнейшей обработке данных. Это возможно в большинстве случаев программирования пользовательского интерфейса, где мы обычно можем проверить ввод неразрешенных данных. Если в текстовом поле могут быть введены только слова «car», «truck» или «bus», очень легко не дать пользователю ввести некорректные данные. Обычным решением такой проблемы будет использование поля со списком. Однако даже здесь могут появиться ошибки. Например, пользователь может ввести день

рождения как 01.01.80 и возраст 30 лет (в 2000 году). Согласованность этих данных легко проверить, однако именно разработчик обязан продумать все возможные проверки согласованности и граничных условий. Эту проверку может быть трудно сделать абсолютной, особенно когда в ней задействованы несколько полей. Когда предоставление данных внешнего объекта выполняется отдельной программой, вероятность ошибки еще больше.

Учитывая, что вероятные ошибки должны быть обработаны, необходимо выяснить, как отдельная программа или метод обрабатывает некорректные входные данные. (Например, метод, рассчитывающий баланс на банковском счете, если в предусловиях метода требуется, чтобы параметр счета был корректным.) Если все этапы процесса программирования были соответствующим образом выполнены, параметры метода всегда будут разрешенными, в какой бы момент метод ни был вызван, но должны ли мы программировать проверку значения параметра, если наш проект или реализация имеют дефекты? Это зависит от требований и обсуждается в конце этого раздела.

Например, предположим, что есть системное требование, согласно которому продолжение выполнения программы превыше всего, даже если выполнение ухудшилось или стало дефективным. В этом случае программист должен рассматривать все возможные входные данные, даже бессмысленные. Например, представьте себе программу, контролирующую работу сердца и снабжение пациента кислородом. Допустим, что мы кодируем метод `process(int measurementType, …)`, где параметр `measurementType` должен быть положительным числом. Будем считать, что зависание программы недопустимо, даже если внутренний метод получил некорректное целочисленное значение параметра в связи с дефектом разработки. Тогда этот метод проверит входные данные и либо, если это возможно, установит по умолчанию безопасные значения, либо переведет всю программу в другой режим работы, либо сгенерирует исключение, передавая вызывающей операции всю ответственность. Программа также должна сгенерировать сигнал тревоги.

Иногда получение неразрешенных данных можно обработать таким способом, который будет согласовываться с детальными требованиями. Это происходит, например, когда данные пересылаются через плохую линию передачи. Получающий метод может быть разработан таким образом, чтобы он ожидал некоторые конкретные данные, однако программа часто требует продолжать выполнение, даже если полученные данные некорректны. В этом случае данные следует проверить, а ошибки обработать в соответствии с требованиями (например, «Если сигнал не лежит в диапазоне от 3,6 до 10,9, сбросьте его значение и ожидайте следующего сигнала»).

Что можно сказать о методах, исключительное поведение которых не определено в требованиях? Во-первых, их предусловия должны быть подробно определены, чтобы условия, при которых они вызываются, были четко сформулированы. Но нужно ли даже в этом случае проверять значения параметров, чтобы убедиться в удовлетворении предусловий? Чтобы выяснить это, мы будем различать выполнение во время разработки и выполнение после ввода в действие.

Выполнение во время разработки позволяет тестировать и проверять код во многих частях программы, и мы, возможно, захотим добавить код, который будет проверять предусловия, как показано ниже.

```
/** предусловие: параметры положительны */
int sum( int int1P, int2P ) {
// код проверки при разработке: проверить параметры
…
// теперь выполнить обработку
… }
```

При рассмотрении выполнения *поставленного заказчику продукта* необходима другая точка зрения. Если метод был вызван с отрицательными параметрами, это показывает, что существует ошибка в самой программе. Мы бы хотели защитить себя от наших собственных ошибок, но «лечение» должно быть предпочтительнее «болезни».

```
/** предусловие: параметры положительны */
int sum( int int1P, int int2P ) {
    // код проверки развернутой программы: проверяем параметры
    // только если нам очевидно, что делать, если параметры отрицательны
    …
    // теперь выполнить обработку
    … }
```

«Болезнь» в этом примере — случаи, когда параметр, который должен быть положительным, является отрицательным. «Лечение» должно вернуть ответ, не согласующийся с состоянием программы.

Без четкой политики такая проверка для выпущенных продуктов является скользкой дорожкой, если проверка всех частей программы невозможна (например, следует ли проверять код проверки?). При фиксированном времени разработки есть опасность потратить его на добавление кода встроенной защиты вместо того, чтобы проверить правильность основы самой системы.

Разработчики теряют контроль над программой, если они используют значения по умолчанию, последствия которых непредсказуемы. Для этого существует несколько причин. Во-первых, неэтично распространять без предупреждения программу, обрабатывающую дефект разработки с частично некорректным продолжением (то есть таким продолжением, которое не предусмотрено требованиями). Дефект — это ошибка, но произвольное значение по умолчанию, не указанное четко в требованиях, покрывает эту ошибку. На практике часто предпочитают перезапустить остановленную программу вместо продолжения ее некорректной работы (представьте себе программу, планирующую маршруты самолетов). Во-вторых, недисциплинированная обработка ошибок скрывает дефект, и обнаружить такой дефект становится дороже, чем позволить зависание программы (с надеждой, что оно произойдет во время тестирования). В-третьих, мы должны быть последовательны в предположениях при проектировании программы. Код, в котором прослеживается принцип: «Я думаю, мы спроектировали это корректно, но если это не так, вот как мы собираемся выполнять программу», не решает проблему разработки и представляет собой плохое программирование.

7.2.6. Другие практические вопросы

- Перед тем как изменять значение переменной, убедитесь, что при чтении значения не произошло никакой ошибки. В Java программисты стараются следовать этому правилу, поскольку в случае неправильного чтения возникает исключение и прочитанное значение считают бесполезным.
- Будьте особенно осторожны при применении множественного наследования в C++. (Java избавлен от этих проблем благодаря запрету на множественное наследование.) Например, когда оба породивших класса имеют переменные с одинаковым именем, порожденный класс имеет две версии переменной, что представляет собой трудную в решении ситуацию.

7.3. Стандарты программирования

Использование стандартов способствует дисциплине программирования и повышает читаемость и переносимость программ. Мы предоставим вашему вниманию набор стандартов, который можно использовать в качестве примера. Некоторые из приведенных далее стандартов были взяты из [4]. Другие можно найти на сайте Sun Corporation, посвященном Java.

7.3.1. Соглашения об именах: примеры на языке Java

Используйте соглашения об именах переменных. Разработчики обычно принимают близко к сердцу свои любимые соглашения, и часто бывает невозможно прийти к единому мнению. И все же соглашения необходимы. На координирование соглашений и способ их выработки следует выделить ограниченное время. Например, можно поручить члену команды разработать черновик соглашений и отправить его по электронной почте остальным членам команды для отзывов. Затем набор соглашений должен быть утвержден лицом, назначенным на разработку соглашений, и лидером команды. Должны быть также сформулированы основные идеи относительно того, в каких случаях возможно отступать от этих соглашений.

Ниже приводится пример соглашения об именовании.

- Именуйте объекты с помощью конкатенации слов, например `длинаЦилиндра`. Такие имена легко понять, и они экономят место. Исключения со знаком подчеркивания оставляются на выбор программиста. Например, вместо `аАААААААвто` было бы лучше написать `а_ААА_АА_А_Авто`.
- Начинайте имена классов с большой буквы. Так вы будете легко отличать их от имен переменных. Некоторые инструменты начинают имена объектов стандартными буквами или комбинациями букв, такими как `С...` для классов, например `СЗаказчик`. Это полезно, когда важность понимания типов имен выше возможной странности написания.

- Имена переменных начинайте с маленькой буквы. Исключением могут быть константы.
- Используйте в именах констант прописные буквы, например `Я_ЕСТЬ_КОНСТАНТА` (используйте спецификаторы `static final`). Имя `ЯЕСТЬКОНСТАНТА` было бы трудно читать; `ЯестьКОНСТАНТА` можно спутать с классом; `яестькон-станта` никак не указывает на то, что это действительно константа.
- Начинайте (или заканчивайте) имена атрибутов классов подчеркиванием, например `_timeOfDay`, чтобы отличать их от других переменных, поскольку они являются глобальными по отношению ко всему объекту. Это соглашение используется [33], но и критикуется [4].

  Существует соглашение, используемое в нашем примере, по которому суффикс `I` является признаком атрибута, например `_timeOfDayI`. Каждый атрибут является глобальным для каждого объекта класса, и когда такое имя встречается в блоке кода, удобно знать, что это атрибут.
- Договоритесь о нотации для выделения статических переменных класса. В нашем примере используется суффикс `S`, например `numCarsEverBuiltsS`. Вспомните, что статическая переменная является глобальной для класса, и поэтому нам полезно знать, что переменная, появившаяся в части кода, является одной из них.
- Для методов доступа используйте имена, начинающиеся с `get`, `set` и `is`, например, `getName()`, `setName()`, `isBox()` (возвращает значение типа `Boolean`).

  В качестве альтернативы можно использовать `name()` и `name( String )` для имени атрибута (например, в CORBA [84]).
- Добавьте стандартные дополнительные методы `get` и `set` для коллекций, например `insertIntoName()`, `removeFromName()`, `newName()`.
- Договоритесь о соглашении для параметров. Можно использовать префикс `a`, например `sum( int aNum1P, int aNum2P )`, или суффикс `P`, например `sum( int Num1P, int Num2P )`.

Документируя методы, описывайте их следующие характеристики:

- предусловия и постусловия (см. обсуждение корректности программы ниже);
- что делает метод;
- почему он делает то, что делает;
- какие параметры следует ему передать (для Javadoc используйте тег `@param`);
- исключения, генерируемые им (для Javadoc используйте тег `@exception`);
- причина выбора видимости (`private` и т. д.);
- способы изменения атрибутов;
- известные ошибки;
- описание теста с указанием, тестировался ли метод, и местоположение тестового сценария;
- история изменений, если вы не используете систему управления конфигурациями;

- пример работы метода;
- специальная документация о потоковых и синхронизированных методах.

Используйте согласованный стандарт для разделителей. Поскольку одиночные пустые строки полезны для разделения частей кода внутри метода, согласованным стандартом могут быть две пустых строки между методами.

Договоритесь о следующих стандартах внутри методов:

- выполняйте только одну операцию на одной строке;
- старайтесь, чтобы ваши методы помещались на один экран;
- используйте скобки внутри выражений, чтобы разъяснить их значение, даже если синтаксис языка этого не требует. Это применение принципа «если знаешь это, покажи».

Присваивая имена классам, используйте слова в единственном числе, например `Заказчик`, за исключением очевидных случаев, когда вы хотите собрать объекты (в этом случае `Заказчики` подошло бы больше). Чтобы избежать размножения классов, иногда желательно, чтобы класс собирал все свои элементы. Это можно сделать с помощью статического элемента данных класса.

7.3.2. Документирование атрибутов

(См. [4].)

- Укажите все подходящие инварианты (вычислимые факты относительно атрибутов, например «36 < _length * _width < 193".)

Для каждого атрибута:

- сформулируйте его цель;
- предоставьте все подходящие инварианты (вычислимые факты относительно атрибутов, например «1 < _age < 130" или »36 < _length * _width < 193".)

Подробное описание инвариантов вы найдете в разделе 7.4.

7.3.3. Константы

Перед тем как отнести переменную к типу `final` (сделать ее константой), убедитесь, что она должна быть таковой. Эмблер [4] предлагает использовать не саму константу, а метод, представляющий ее. Например, вместо

```
protected static final MAX_CHARS_IN_NAME = 20;
```

следует использовать метод:

```
protected final static int getMaxCharsInName()
{    return 20;
}
```

Этот способ выигрывает благодаря своей наглядности (например, при использовании метода `getMaxCharsInName()` вместо непонятного числа `20`) и гибкости.

7.3.4. Инициализация атрибутов

Атрибуты всегда должны быть инициализированы, чтобы программист мог контролировать свою программу. Обычно мы представляем инициализацию следующим образом:

```
private float balanceI = 0; //соглашение: атрибут xxxI
```

Однако атрибут, нуждающийся в инициализации, может быть объектом другого класса:

```
private Customer customerI;
```

Эта инициализация часто выполняется с помощью конструктора:

```
private Customer customerI = new Customer( «Edward», «Jones» );
```

Проблема этой техники заключается, согласно [4], в ее эксплуатационной надежности. Когда в **Customer** будут добавляться новые атрибуты, все эти инициализации, возможно, придется обновлять. Другими словами, она нарушает объектно-ориентированную цель инкапсуляции всех связанных с **Customer** аспектов класса **Customer**. Другая проблема заключается в том, что такая конструкция может потребовать ненужного постоянного хранения.

Согласованное решение заключается в использовании инициализации только в тех случаях, когда к значению сначала получается доступ («ленивая инициализация»). В общем случае мы добавляем в **MyClass** статическую функцию, которую назовем **getDefaultMyClass()**. Теперь можно объявлять атрибуты в других классах типа **MyClass** без инициализации и менять их значения только при первом доступе. Пример представлен в листинге 7.1, где **MyClass** — это класс **Customer**.

Листинг 7.1. Одно из решений инициализации объекта

```
Используйте инициализацию при первом обращении к значению.
Добавьте в MyClass статическую функцию getDefaultMyClass().
Атрибуты объявляются без инициализации, и им присваиваются значения при первом
обращении.

В классе Customer:
```


```
...
Public static Customer getDefaultCustomer()
// Далее по умолчанию выбраны разумные значения
{       return new Customer
            ( «John», «Doe», 0, 1000, -2000 );
}

private float balanceI = -10;
private Customer customerI;
```

```
public Account ( . . . ) . . .

public float getBalance()
{   return balanceI;
}

public Customer getCustomer()                          //Обращение к customerI
{
    if( customerI == null )                             //Обращения не было
        customerI = Customer.getDefaultCustomer();      //Исходное значение
    return customerI;                                   //Текущее значение
}

public getDefaultAccount()       //Для пользователей Account
{  return new Account( -10. 3. «regular» );
}
```

7.4. Доказуемо корректные программы

Говорят, что программма *доказуемо корректна*, если *имеется математическое доказательство* того, что она удовлетворяет сформулированным применительно к ней требованиям. Доказательство основывается на этих требованиях и на коде программы. Оно никак не связано ни с результатами компиляции, ни с результатами тестирования программы. Это идеальный случай.

Хорошее математическое доказательство — это правильно построенное рассуждение об истинности доказываемого утверждения, убедительное как для самого автора доказательства, так и для всех прочих. Ключевым моментом в проведении доказательств является использование языка математики, однако чтобы использовать этот язык, прежде всего нужно, чтобы само доказываемое утверждение было сформулировано математически.

Во-первых, требование само по себе должно быть сформулировано точно. В главе 4 мы описали Z-спецификации для формального изложения требований. Здесь мы будем использовать более простую форму введенных в главе 4 требований, — *предусловия и постусловия*. Предусловия определяют все предположения, сделанные при вызове функции. Постусловия определяют необходимое состояние по завершении выполнения функции. Например, рассмотрим следующее требование к функции *f()*:

Предусловие: g — массив целых чисел длины n (g «дано»).
Постусловие: $r = \max\{ g[0], g[1], ..., g[n-1] \}$ (r — «результат»).

В неформальных терминах, это требование вызывает функцию *f()* для определения максимального значения массива g. Это легко запрограммировать, однако мы сделаем это так, чтобы нам было легко доказать корректность кода.

Одним из способов познакомиться с программированием фрагментов кода, корректность которых можно доказать, является применение (псевдокода):

```
while (Переменным присвоены неразрешенные значения)
        Выполнить соответствующие действия
```

```
/*Осталось только доказать, что этот цикл конечный, чтобы убедиться,
что переменным будут присвоены необходимые значения*/
```

Для доказательства того, что переменной присваивается необходимое значение, программисту достаточно убедиться в конечности цикла. Если программист может доказать, что цикл конечен, утверждение «Переменной не присваивается необходимое значение» не будет справедливым (двойное отрицание), и необходимое состояние будет достигнуто.

На первый взгляд, доказуемо корректные программы кажутся новым направлением в программировании. Это не совсем так. Интуитивный процесс, обычно используемый программистом, по сути ничем не отличается от формального доказательства: формальность просто помогает контролировать процесс с гораздо более высокой точностью. Например, большинство из нас будут программировать функцию *max()* посредством создания цикла и накопления максимального значения «из рассмотренных». Другими словами, во время компиляции мы будем хранить следующее утверждение как истинное:

[$j <= n-1$] AND [r = maximum{ $g[0], g[1], ..., g[j]$ }] //утверждение I

Как упоминалось в предыдущих главах, подобное этому утверждение, которое должно поддерживаться истинным, называется *инвариантом*. Мы также говорим, что утверждение *сохраняется инвариантным* (неизменным). Хотя значения отдельных переменных в нем (j и r) изменяются, *утверждение в целом остается истинным*. По ходу вычисления *max()* и роста j значения j и r согласовываются, благодаря чему утверждение I остается истинным. Обратите внимание, что *max()* — это имя функции, которую мы разрабатываем, в то время как термин «максимум», использовавшийся в утверждении инварианта, является математическим понятием, с которым читатель должен быть знаком.

Инвариант используется в качестве фундамента, на котором строится вся программа. Аналогичную роль играет инфраструктура в структурном программировании: «дано», на котором мы можем основываться, чтобы удовлетворить требованиям. Это проиллюстрировано на рис. 7.8, где показана разработанная инфраструктура, предоставляющая «инвариант», на основе которого можно строить горное шоссе.

Программа с доказуемой корректностью для *max()* представлена в листинге 7.2.

Листинг 7.2. Программа с доказуемой корректностью, определяющая максимум в массиве

```
// Определить I: 0 =< j <= n-1 < 100 & r = max{g[0], g[1], ..., g[j]}
// После следующих двух команд I будет истинно:
int r=g[0];
int j=0;

// В этом блоке для I поддерживается значение «истина»:
while( j < n-1 )
{    if( g[j+1] > r )
           r=g[j+1];
     ++j;
}
```

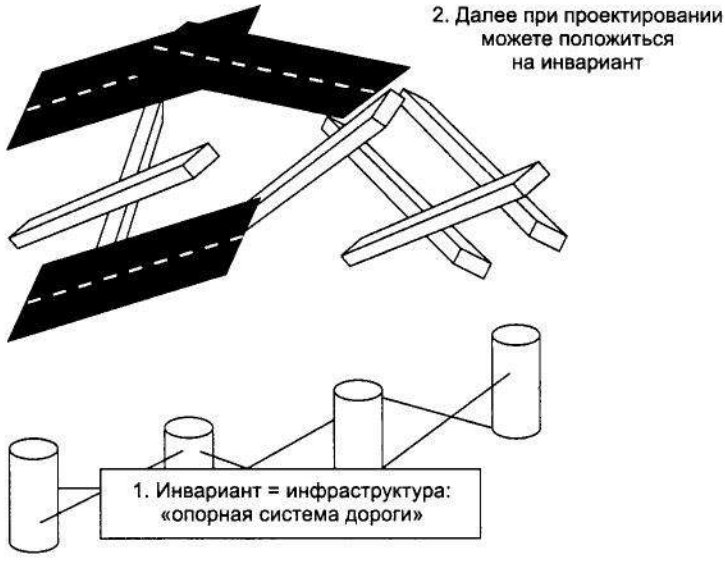


Рис. 7.8. Инвариант как инфраструктура

Обратите внимание, что мы используем цикл `while` вместо `for`, поскольку циклы `while` проще в использовании при доказательстве корректности. Существует ограничение по *n*, поскольку реальные компьютеры не могут гарантировать эффективность программы для любого *n* без ограничения. Доказательство того, что программа корректна, сформулировано в листинге 7.3.

Листинг 7.3. Программа с доказуемой корректностью, определяющая максимум в массиве: доказательство корректности

```
/* Предположив, что цикл конечен (будет доказано ниже), мы знаем на этот момент,
что j < n-1 больше не истинно. Помимо этого, мы сохранили I как инвариант
(истина). Объединяя эти утверждения, получаем:
j < n-1 ложь, AND
j <= n-1 (из первого утверждения), AND
r = max {g[0], g[1], …, g[j]} (из первого утверждения)
Следовательно, j = n-1 AND r = max {g[0], g[1], …, g[n-1]}
Что и требовалось доказать.
Остается только доказать, что цикл while конечен. Поскольку величина I
сохранялась инвариантной, величина n-j всегда положительна. Более того, n-j
уменьшается на 1 с каждой итерацией. Единственный случай, когда такая ситуация
будет иметь место, – когда цикл конечен.*/
```

Этот простой пример иллюстрирует мнение ученых и программистов, полагающих, что технология программирования со временем будет все активнее использовать формальные методы. Они утверждают, что формальные методы постепенно делают наш процесс реализации программы более профессиональным и точным. В книге [25] показано, что даже простейшие программы могут обладать богатством потенциальных изменений, более или менее эффективных,

удовлетворяющих требованиям или нет. Таким образом, не удивительно, что в коде даже лучших программистов скрываются ошибки. Следовательно, нам нужны все возможные инструменты и техники для производства надежных программ.

Приведенный ниже пример взят из [25]. Более простой для понимания является книга [39]. Задача состоит в нахождении наибольшего общего делителя (НОД) двух положительных целых чисел: наибольшего натурального числа, на которое делятся оба заданных числа. Например, НОД для 18 и 12 равен 6.

Мы будем различать «НОД» — речевой термин (см. предыдущий раздел), и *gcd()* — код, который мы будем создавать для реализации НОД. Программа (листинг 7.4) использует тот математический факт, что НОД двух положительных целых чисел *a* и *b* тот же, что и у пары *a–b* и *b* при условии, что разность *a–b* положительна. Например, НОД для 30 и 18 (равный 6) будет тем же, что и НОД для (30–18) и 18. В последней паре чисел, 12 и 18, НОД также равен 6.

Требования для метода — `gcd(int xP, int yP)` — можно сформулировать в терминах предусловий и постусловий.

- Предусловия:

  xP и *yP* — натуральные числа.

- Постусловия (повторить определение НОД):

  1) *gcd(xP, yP)* — положительное целое;

  2) *gcd(xP, yP)* кратно *xP* и *yP*;

  3) если *z* натуральное число, кратное *xP* и *yP*, тогда *z* <= *gcd(xP, yP)*.

Код для функции `gcd(int xP, int yP)` показан в листингах 7.4 и 7.5. Доказательство окончания программы — это одно из нескольких возможных объяснений причин, по которым цикл будет иметь конечное число итераций.

Листинг 7.4. Программа отыскания НОД с доказуемой корректностью кода

```
// Пусть I - следующее утверждение: gcd(xP, yP) = gcd(x, y)

// Инициализировать переменные, чтобы I было истинно:
int x = xP;
int y = yP;

/* Следующий цикл сохраняет I и в результате дает x==y.
Когда цикл закончится (доказательство конечности цикла см. в листинге 7.5):
    НОД(xP, yP) = НОД(x, y)            согласно инварианту
        = НОД (x, x)                   поскольку  x==y
        = x
Вернуть x как результат, и наша работа на этом закончена.*/
```

Листинг 7.5. Программа отыскания НОД с доказуемой корректностью кода: доказательство конечности цикла

```
while( !( x==y ) )
{     if( x>y )
```

```
            x = x - y;     // I сохраняется (свойство НОД)
        else               // !(x==y) AND !(x>y), значит, x < y
            y = y - x;     // I сохраняется (то же самое свойство НОД)
}
/* Величина x + y уменьшается по крайней мере на 1 на каждой итерации, но
остается положительной. Значит, число итераций должно быть конечно.*/
System.out.println(«НОД чисел» + xP + «и» + yP «равен» + x);
```

Подведем итоги обсуждения программ с доказуемой корректностью кода. В общем случае создание программ с доказуемой корректностью подразумевает следующие шаги.

1. Определите отношение *I* среди переменных, которое легко установить, и отношение *r* так, что *I* AND *r* является постусловием.
2. Напишите код, в котором *I* будет истинно.
3. Создайте код со следующей структурой:

```
while(!r)
{    Выполнение операций, которые
    сохраняют I инвариантом и
    в конце концов приведут к окончанию цикла.
}
```

4. Докажите, что цикл `while` конечен.

7.5. Инструменты и среды программирования

Часто говорят, что человек сам создает инструменты, и это также относится к разработчикам программ. Существует постоянно растущее число инструментов, помогающих в программировании.

Интерактивные среды разработки широко используются, помогая программистам производить больше кода за меньшие сроки. Сюда относятся возможность перетаскивания для конструирования компонентов графического пользовательского интерфейса, графическое представление каталогов, отладчики, мастера и т. д. В Java Beans используется подход, стандартизирующий исходный текст таким образом, что среда разработки Java Beans может обрабатывать исходные данные. Преимущество этого подхода в том, что разработчики не привязаны к одной среде разработки. Значительно упростилось создание COM-объектов.

Программы профилирования, такие как *Jprobe*, могут использоваться для накопления статистических данных, например:

- совокупное время центрального процессора и истекшее время;
- время, затраченное на каждый метод;
- совокупное количество созданных объектов;

- количество вызовов;
- среднее время работы каждого метода.

Программы должны иметь исходный код в той или иной форме. Многие организации первого уровня CMM, напротив, не производят ничего, кроме исходного кода. После того как производители инструментов осознали, что их клиенты будут сами производить код, возможно документированный, они стали разрабатывать средства обратного проектирования, получающие на вход исходный код и формирующие документацию. Примером инструмента обратного проектирования исходного кода является Javadoc (см. раздел 6.7.3). Обратное проектирование обсуждается более подробно в главе 9.

Несколько объектно-ориентированных инструментов (таких, как Rational Rose, Together/J/C++) генерируют исходный код из объектных моделей. Эти инструменты прямого проектирования не могут сгенерировать нечто большее, чем каркас кода, по которому должна работать программа, являющаяся финальной реализацией. Те же инструменты также выполняют обратное проектирование, механически производя объектные модели из исходного кода (отсюда появился термин «проектирование туда-обратно»).

Эволюция инструментальных средств в других областях проектирования (например, CAD/CAM) позволяет надеяться, что инструменты программирования будут продолжать улучшаться, будут так же активно использоваться программистами и приведут к снижению количества монотонных и механических заданий.

7.6. Качество в реализации

В этом разделе обсуждаются метрики и методы достижения качественных реализаций. Ниже представлены контрольные списки инспектирования кода.

ОДИН ИЗ СПОСОБОВ ИНСПЕКТИРОВАНИЯ КОДА (1): КЛАССЫ В ЦЕЛОМ

1. Подходит ли имя для класса:
   - согласуется с требованиями или проектом?
   - достаточно конкретно (обобщенно)?
2. Может ли класс быть абстрактным (чтобы использоваться только как основа)?
3. Описывает ли заголовок класса его цель?
4. Ссылается ли заголовок класса на требования элемента проекта, которому этот класс соответствует?
5. Указан ли там пакет, к которому этот класс принадлежит?
6. Является ли класс настолько скрытым, насколько это возможно?
7. Не сделать ли его `final` (Java)?
8. Использовались ли стандарты документации? Например, Javadoc.

ОДИН ИЗ СПОСОБОВ ИНСПЕКТИРОВАНИЯ КОДА (2): АТРИБУТЫ

1. Является ли атрибут действительно необходимым?
2. Можно ли сделать его статическим?
   - Нужна ли в каждом классе своя переменная?
3. Не сделать ли его `final`?
   - На самом ли деле изменяется его значение?
     - Будет ли достаточно одного метода `get`?
4. Правильно ли были применены соглашения о присваивании имен?
5. Является ли атрибут настолько скрытым, насколько это возможно?
6. Являются ли атрибуты настолько независимыми, насколько это возможно?
7. Существует ли ясная стратегия инициализации
   - в момент объявления?
   - с конструктором (конструкторами)?
   - используя `static{}`?
   - все вышеупомянутое? Как?

ОДИН ИЗ СПОСОБОВ ИНСПЕКТИРОВАНИЯ КОДА (3): КОНСТРУКТОРЫ

1. Является ли конструктор необходимым?
   - Не лучше ли использовать factory-метод?
     - Большая гибкость.
     - Дополнительная функциональность.
2. Нет ли уже существующих подобных конструкторов? (Только для Java.)
3. Все ли атрибуты инициализирует конструктор?
4. Является ли конструктор настолько скрытым, насколько это возможно?
5. Исполняет ли конструктор при необходимости унаследованные конструкторы?

ОДИН ИЗ СПОСОБОВ ИНСПЕКТИРОВАНИЯ КОДА (4): ЗАГОЛОВКИ МЕТОДОВ

1. Присвоено ли методу подходящее имя?
   - Согласуется ли имя метода с требованиями или проектом?
2. Является ли метод настолько скрытым, насколько это возможно?
3. Может ли он быть `static`?
4. Может ли он быть `final`?
5. Описана ли в заголовке цель метода?
6. Ссылается ли заголовок на раздел в требованиях и (или) проекте, которому он соответствует?
7. Формулирует ли он все необходимые варианты (раздел 7.4)?
8. Сформулированы ли в методе все предусловия?
9. Сформулированы ли в методе все постусловия?
10. Использованы ли в методе стандарты документации?
11. Ограничены ли типы параметров (раздел 7.2.5)?

ОДИН ИЗ СПОСОБОВ ИНСПЕКТИРОВАНИЯ КОДА (5): МЕТОДЫ

1. Согласуется ли алгоритм метода с псевдокодом или блок-схемой в детальном проектировании?
2. Выполняет ли код сформулированные предусловия и только их?
3. Выполняет ли код абсолютно все постусловия?
4. Выполняет ли код требуемый инвариант?
5. Каждый ли цикл конечен?
6. Выполнены ли необходимые стандарты нотации?
7. Каждая ли строка была внимательно проверена?
8. Все ли фигурные скобки сбалансированы?
9. Учтены ли неразрешенные значения параметров (раздел 7.2.5)?
10. Возвращает ли код корректный тип?
11. Добавлены ли к коду подробные комментарии?

7.6.1. Стандартные метрики для исходного кода

7.6.1.1. Подсчет строк

Число строк кода хоть и не является идеальным показателем, но представляет собой одну из важных программных метрик. Необходимо выбрать стандартный способ подсчета строк. Например, нужно решить, как учитывать при подсчете:

- утверждения, занимающие несколько строк (1 или *n*?);
- комментарии (0?);
- строки, состоящие из `while`, `for`, `do` и т. д. (1?).

Выбор может зависеть от применяемого стандарта и от используемого инструментария, осуществляющего автоматический подсчет. Следование выбранному способу подсчета значительнее важнее, чем сам выбор этого способа.

7.6.1.2. Метрики IEEE

Ниже приведен небольшой пример метрик из [59], показывающий варианты оценок исходного кода. Эти оценки не зависят от тестирования (оно будет рассмотрено в следующей главе).

IEEE-метрика 14. Программные научные метрики

Используемые измеряемые величины:

Пусть n_1 — это число отдельных операторов (+, * и т. д.) в программе. Например, в программе {x = x + y; z = x * w} $n_1 = 2$.

Пусть n_2 = число отдельных операндов в программе. Например, в программе {x = x + y; z = x * w} $n_2 = 4$, поскольку операторы сложения и умножения имеют по два операнда каждый.

Пусть N_1 = число всех операторов в программе.

Пусть N_2 = число всех операндов в программе.

Пример оценки:
предполагаемая длина программы: $n_1(\log n_1) + n_2(\log n_2)$;
сложность программы: $(n_1 N_2) / (2n_2)$.
Дополнительную информацию по программным научным метрикам см. в [53].

IEEE-метрика 16. Цикломатическая сложность

Эта метрика определяет структурную сложность блока кода по вычислительным циклам, которые являются важными факторами сложности. Ее можно использовать для определения модулей, сложность которых необходимо уменьшить.

Примером политики, основанной на этой метрике, является требование специально проверить все модули с цикломатической сложностью выше средней на 20 %.

Используемые измеряемые величины:
N — число узлов (операторов программы);
E — число ребер (ребро соединяет узел m с узлом n, если оператор n следует сразу за оператором m).

Один из способов вычисления цикломатической сложности заключается в вычислении $E - N + 1$. Другой способ получения этой оценки заключается в вычислении количества замкнутых областей, образованных ребрами, которые нельзя разбить на меньшие области. Это очевидно из рис. 7.9.

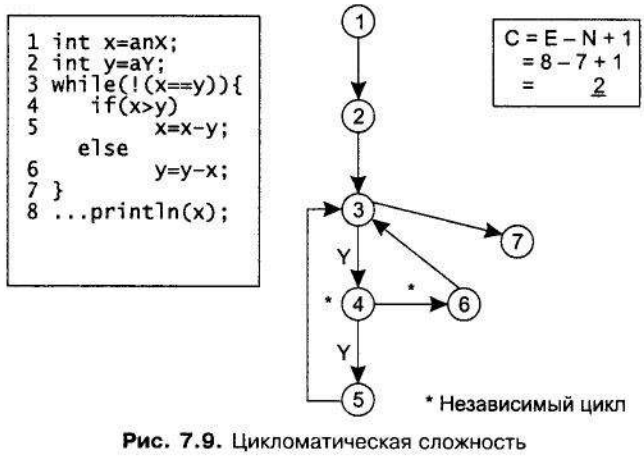


Рис. 7.9. Цикломатическая сложность

7.6.2. Индивидуальные метрики для исходного кода

Цикломатическая сложность выражает число циклов в программе, но не проводит различия между вложенными и невложенными циклами. В общем случае циклы внутри циклов подвержены ошибкам чаще, чем последовательности независимых циклов.

Нетрудно создать метрики, измеряющие важные величины. Например, чтобы вычислить сложность цикла, можно предположить, что каждый вложенный цикл увеличивает сложность на порядок (грубо говоря, в 10 раз). Приведенный ниже пример можно расценивать как описание коэффициента сложности вложенных циклов.

Цикл 1 — сложность 1.

Цикл 2 — сложность 1.

Цикл 2.1 — сложность 10 (цикл внутри цикла 2).

Цикл 2.2 — сложность 10.

Цикл 2.2.1 — сложность 100.

Цикл 3 — сложность 1.

Цикл 4 — сложность 1.

Цикл 5 — сложность 1.

Цикл 5.1 — сложность 10.

Используя эту метрику, получаем сложность 135. Эту метрику можно нормировать, разделив полученную величину на число строк кода и сравнив результат со средним значением по фирме.

7.6.3. Инспектирование кода

Вспомните, что *серьезность ошибок* является важной составляющей инспектирования данных, поскольку именно она позволяет расставлять приоритеты дефектов и тем самым рационально распределять работу (табл. 7.1). Нет смысла тратить время инспектирования на тривиальные дефекты. Неплохим способом обработки тривиальных дефектов является ситуация, когда инспекторы передают распечатку исходного кода с замечаниями автору.

Таблица 7.1. Классификация серьезности дефектов с использованием метода отбраковки

Серьезность дефекта	Описание дефекта
Важный	Требование не выполнено
Средний	Дефект нельзя отнести ни к важным, ни к тривиальным
Тривиальный	Дефект не влияет на работу программы и сопровождение

Кроме того, дефекты можно описать, основываясь на их *типе*; одна из таких классификаций, приведенная в главе 6, повторена ниже.

- Проблема логики (забытые случаи или шаги; повторенная логика; пропуск граничных условий; неважные функции; некорректное прерывание; недостаток текста условия; проверка неправильных переменных; некорректность итерационных циклов и т. д.).

- Вычисления (некорректное или недостаточное уравнение; потеря точности; ошибка в знаке).
- Интерфейс и синхронизация (некорректная обработка прерываний; некорректная синхронизация ввода-вывода, несоответствие модулей или подпрограмм).
- Обработка данных (некорректная инициализация данных; некорректное получение или хранение данных; некорректный масштаб или единицы измерения данных)
- Границы данных.
- Данные (некорректны или отсутствуют данные с датчиков, данные оператора, данные таблиц, внешние данные, выходные данные, входные данные).
- Документация (неоднозначное описание и т. д.).
- Качество документа (несоответствие применяемым стандартам и т. д.).
- Расширение (изменение в требованиях и т. д.).
- Ошибки, возникшие в результате предыдущих исправлений.
- Возможность взаимодействия (несовместимость с другими программами или компонентами).
- Соответствие стандартам.
- Другие (не перечисленные выше).
- В качестве примера мы проведем инспектирование кода метода `adjustQuality()` класса *ПерсонажВстречи* в нашем учебном проекте видеоигры (рис. 7.10 и рис. 7.11).

```
/**
*3.2.ПИ.3.1. Настройка значений характеристик персонажа игрока*/
public synchronized void adjustQuality
  (String qualityP,float qualityValueP)
    {
     float originalSumM = sumOfQualitiesQ; //должна быть неизменной

     try
     {//pc IF qualityP не распознана
     //pc записать ошибку в log-файл и сообщить
     //pc пользователю, что характеристики не были изменены
     //Необходимо текущее значение qualityP
     float originalValueOfaQualityPM = qualValueP[indexOf(qualityP)];

     //pc IF qualityValueP лежит вне предусмотренных пределов
     //pc    записать ошибку в log-файл и сообщить пользователю,
     //pc    что характеристики не были изменены...
     //pc ELSE
     //pc    Присвоить заданной характеристике значение qualityValueP
     setQuality(qualityP, qualityValueP);
```

Псевдокод не приведен; серьезность - средняя; тип - данные

Дефект тот же, что выше

Рис. 7.10. Исходный код adjustQuality() и обнаруженные дефекты (1)

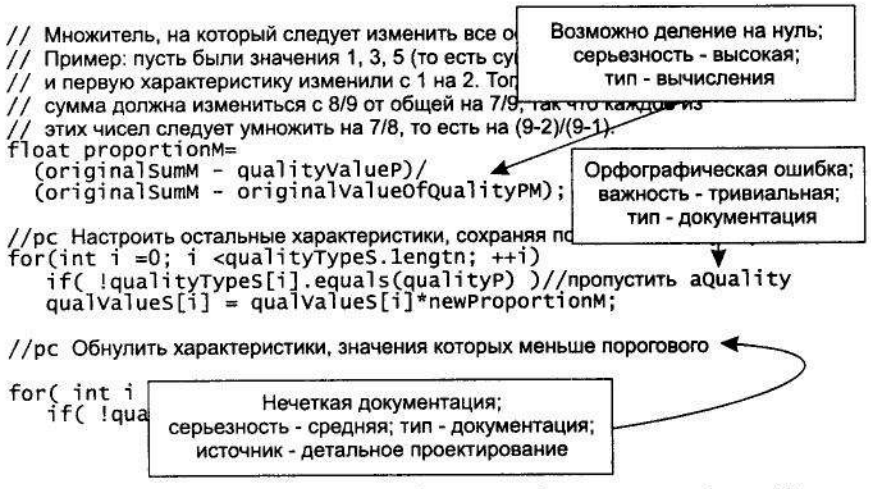


Рис. 7.11. Исходный код adjustQuality() и обнаруженные дефекты (2)

7.6.4. Индивидуальная программная документация (PSD)

Каждый разработчик обязан хранить документацию своей текущей работы. Эта документация называется *индивидуальной программной документацией* (PSD — Personal Software Documentation). Она позволяет в любой момент получить информацию о статусе разработки, и часть ее позднее будет добавлена в архив проекта. Эта документация может содержать исходный код и данные об индивидуальной регистрации дефектов, состоящие из следующих пунктов.

- Типы дефектов.
- Индивидуальный этап, на котором дефект возник.
- Индивидуальный этап, на котором дефект был исправлен.

  Индивидуальные этапы — это:

  1) дополнительное детальное проектирование (если применимо);
  2) код (записывайте возникшие или обнаруженные дефекты в исходном коде до компилирования);
  3) отчет о компиляции (записывайте обнаруженные и исправленные дефекты после попытки компиляции);
  4) модульное тестирование. Модульное тестирование проводится группой контроля качества и не является частью этой документации.
- Учет времени. Время, затраченное на дополнительное детальное проектирование, кодирование, компиляцию и тестирование.
- Записная книга проектирования. Содержит статус дополнительного детального проектирования (если применяется) и кода, а также сведения о инцидентах и требующих внимания вопросах разработки.

В индивидуальном процессе разработки, созданном в Институте технологий разработки программного обеспечения (SEI), ведется учет времени и информации о дефектах. Команда или лидер проекта определяет, как использовать и архивировать индивидуальную документацию членов команды. Слово «индивидуальная» не должно восприниматься как определение собственности разработчика: работа, оплачиваемая организацией, обычно является собственностью этой организации. Работу членов команды обычно оценивают по конечным результатам их личного участия в разработке и по тому, насколько хорошо их процесс работы был документирован. Вообще говоря, их работу не оценивают на каждой стадии процесса.

7.7. Итоги процесса реализации

Целью реализации является корректное программирование детального проекта при поддержке эксплуатационной надежности продукта. Использование стандартов программирования помогает в этом процессе. Очень важно иметь организованный подход при определении цели каждого метода, и этого можно достичь путем формулирования предусловий и постусловий. Чтобы корректность кода, особенно потенциально сложного кода, можно было доказать, необходимо писать текст программы организованным способом, желательно допускающим формальное доказательство. Это облегчает доказательство удовлетворения требованиям. Подведем итоги.

- Всегда помните о цели программирования:
  - корректность;
  - четкость.
- Используйте стандарты программирования.
- Определите предусловия и постусловия.
- Докажите корректность отдельных частей без компилятора.
- Реализуйте программу за заранее заданное время.
- Поддерживайте качество и профессионализм.

Упражнения

Ответы и подсказки для упражнений, помеченных символами «о» или «п», приводятся в конце этой главы

Вопросы для проверки

П7.1°. Вы собираетесь закодировать метод. Какие два основных ресурса предоставляют вам информацию о том, что ваш метод должен делать?

П7.2°. Сформулируйте 3–5 шагов, которые обычно происходят после того, как вы закодировали метод.

Для ответов на вопросы 7.3–7.6 обратитесь к разделам 7.2.1–7.2.4.

П7.3. Назовите 3–4 общих принципа выполнения качественной реализации.

П7.4. Дайте 2–3 общих принципа работы с указателями.

П7.5. Дайте 2–3 общих принципа работы с функциями.

П7.6. Дайте 4–6 общих принципов работы с исключениями.

П7.7°. Предположим, вы хотите, чтобы ваша часть программы осуществляла состояние S с использованием переменных типа `float x, y, z`, например `x = max( y, z )`.

1. Покажите общую структуру цикла `while`, выполняющего эту задачу, с доказуемой корректностью.
2. Что необходимо для полноты доказательства?
3. Каким образом структура `while` предоставляет возможность обработки эффективности процедуры?
4. Фрагменты кода не существуют в вакууме. Цикл `while` обычно находится среди другого кода. Опишите в общих чертах способ связи цикла с окружением, который часто используется для установления корректности.

П7.8°. Подведите итог общему строгому способу определения того, что метод должен делать.

П7.9. Назовите 5 метрик исходного кода (ответ ищите в разделе 7.6.1).

Общие упражнения

О7.1. Приведите 2–4 достоинства и 1–2 недостатка принудительного использования стандартов кодирования.

Упражнения в команде

К7.1. («Реализация».)

Реализуйте ключевые части вашей программы в виде прототипа.

Ответы

П7.1. Метод либо соответствует напрямую D-требованию в SRS, либо необходим в проекте.

П7.2. Вы должны измерять время, которое ушло на выполнение работы, как в PSP . Вы должны внимательно инспектировать и тестировать себя. Метод сам по себе либо вместе с другими методами необходимо регулярно инспектировать. Далее метод инспектируется при модульном тестировании и только после этого, если он проходит тестирование успешно, он интегрируется в программу.

П7.7.

1)

```
while (not S)
        Выполнить вычисления
```

2) Докажите конечность цикла.

3) Вероятно, вы можете сравнить эффективность таких алгоритмов, сравнивая число итераций и количество вычислений, выполняемых в теле цикла `while` до окончания этого цикла.

4) Часто существует утверждение (в отношении переменных), остающееся неизменным (инвариантным).

П7.8. Определите предусловия и постусловия.

Пример 1. Обновление Плана контроля качества (SQAP)

(Добавление в раздел: 5.2 примера 2 в главе 1.)

5.2.1. Соглашения по программированию (этот раздел был добавлен)

Будут использоваться приведенные ниже соглашения.

Части имен элементов программы, не являющихся константами, выделяются с помощью прописных букв, например `этоЕстьИмя`.

Имена классов и интерфейсов начинаются с прописной буквы, например `Account`.

Имена атрибутов в классах начинаются со строчной буквы и заканчиваются буквой *I*, например `balanceI`.

Статические переменные (класса) начинаются со строчной буквы и заканчиваются буквой *S*, например `interestRateS`.

Переменные, определенные в методе и являющиеся для метода глобальными, начинаются со строчной буквы и заканчиваются буквой *M*, например `interestM`.

Параметры начинаются со строчной буквы и заканчиваются буквой *P*, например `principalP`.

Переменные `final` будут записываться прописными буквами и знаками подчеркивания, например `BANK_NAME`.

5.2.2. Нотация, показывающая местоположение файлов

Для описания реализации мы будем использовать UML [15].

5.2.3. Индивидуальная программная документация (PSD)

Каждый разработчик будет поддерживать документацию своей работы, которая будет называться *Индивидуальной программной документацией* (PSD). Это позволит разработчику в любой момент сообщать состояние свой работы, и эта документация становится частью архива проекта. Команда или руководитель проекта будет определять способ организации PSD членов команды. Обычно набор лич-

ных документов соответствует задаче, выделенной конкретному разработчику, и состоит из набора классов.

[Примечание для студентов. Пример PSD представлен в разделе «Пример 3» ниже.]

Пример 2. Обновление Плана контроля качества (SQAP). Приложение: Модель реализации

Пример

Часть модели реализации видеоигры *Встреча* приведена на рис. 7.12.

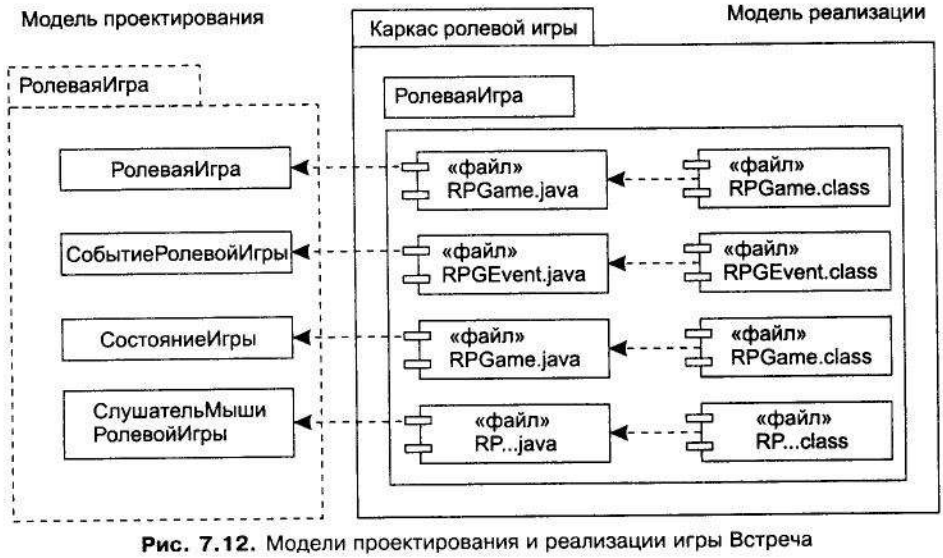


Рис. 7.12. Модели проектирования и реализации игры Встреча

Пример 3. Индивидуальная программная документация (PSD), часть 1

[Примечание для студентов. Этот документ поддерживается каждым разработчиком и описывает текущее состояние работы. Он должен быть достаточно полным, чтобы позволять разработчику давать отчет о статусе выполения на собраниях и при необходимости позволять другим разработчикам принять работу за приемлемое время.

Если разработчик придерживается плана PSP или TSP, формат и содержание этого документа должны быть определены в этих процессах. Формат, приведенный ниже, использует идеи PSP, но не в полной форме. Вторая часть приведена в главе 8.]

1. Введение

Этот документ описывает работу Джона Джонса над классом EncounterCharacter (*ПерсонажВстречи*). Конфигурация этого класса описана в файле PSD _EncounterCharacter. Файлы, на которые мы будем ссылаться, хранятся в каталоге Encounter\PSD\Jjones в системе Galaxy.

2. Учет дефектов

Журнал учетных записей дефектов (табл. 7.2) поддерживается в файле defectLog.

Таблица 7.2. Учетные записи дефектов (Хэмфри)

Дата	Номер	Тип	Этап, на котором обнаружили	Этап, на котором исправили	Время исправления (минуты)
14.06.99	142	Интерфейс	Индивидуальное детальное проектирование	Индивидуальная проверка кода	10
Описание: пропущена проверка длины имени в классе *ПерсонажВстречи*					
16.06.99	143	Документация	Код	Индивидуальная проверка кода	4
Описание: некорректное описание Javadoc в классе *ПерсонажВстречи*					
...	...	...	...	...	...
Эта таблица содержит дефекты, обнаруженные во время модульного тестирования (глава 8)					

3. Учет времени

[Примечание для студентов. Разработчики ведут учет того, сколько времени у них ушло на выполнение различных действий, необходимых в разработке (табл. 7.3). Эта информация очень важна для проекта, поэтому разработчику также предоставляется профессиональный набор инструментов. Информация о затраченном времени может быть собрана как в письменной форме, так и с помощью программных инструментов, резидентно установленных на компьютерах. Инженеры должны разработать общее понимание степени точности, требуемое организацией. Обратите внимание, что приблизительное измерение времени может легко привести к рассогласованиям на практике.]

Эта информация (табл. 7.3) хранится в файле Time_Rrecording_Log.

Таблица 7.3. Учетные записи времени (Хэмфри)

Дата	Начало	Конец	Перерывы	Затраченное время	Этап	Комментарии
06.99	10:04	10:25	4 + 6	11	Детальное проектирование	Проконсультировался у N.
06.99	13:20	16:34	15 + 20	159	Личная проверка кода	Дефект 14
07.99	...					

4. Модульное тестирование. PSD для класса EncounterCharacter (ПерсонажВстречи)

Пример в конце главы 8.

Пример 4. Исходный код (без тестового кода): класс EncounterCharacter (ПерсонажВстречи)

Исходный код класса EncounterCharacter (*ПерсонажВстречи*) представлен в листинге 7.6. Символами //pc обозначается ссылка на псевдокод.

Листинг 7.6. Класс EncounterCharacter (ПерсонажВстречи): исходный код

```
Package Encounter.EncounterCharacters;
/* Имя класса                          EncounterCharacter
* Дата                                01.13.2000
* Уведомление об авторском праве      copyright 1999-2000 by E. Braude
*/
import java.awt.*;
import java.io.*;
import FrameworkRPG.Characters.*;
import TestUtilities.*;

/** Основной класс для персонажей игры Встреча. SDD-ссылка: 6.2.1
* <p> Инварианты: значения qualValueI[] >= 0
* @author Eric Braude, Tom VanCourt
* @version 0.2
*/
public class EncounterCharacter extends GameCharacer
{
/**Суммарное значение характеристик при инициализации*/
private static final float QUAL_TOTAL_INIT = 100.0f;

//Обозначения, используемые при обращении других классов к характеристикам

/**Обозначение одной из характеристик персонажа*/
public static final String QUAL_CONCENTRATION = «concentration»

/**Обозначение одной из характеристик персонажа*/
public static final String QUAL_INTELLIGENCE = «intelligence»

/**Обозначение одной из характеристик персонажа*/
public static final String QUAL_PATIENCE = «patience»

/**Обозначение одной из характеристик персонажа*/
public static final String QUAL_STAMINA = «stamina»
```

```
/**Обозначение одной из характеристик персонажа*/
public static final String QUAL_STRENGTH = «strength»

/**Характеристики, имеющиеся у каждого персонажа <p>Req:3.2.EC.1.1*/
public static final String[] qalityTypeS=
{   QUAL_CONCENTRATION, QUAL_STAMINA,
    QUAL_INTELLIGENCE, QUAL_PATIENCE,
    QUAL_STRENGTH
};

/*Объектные переменные*/
/**Значения характеристик <p> Требование 3.2.ПВ.1.2*/
private float[] qualValueI = new float[qualityTypeS.length];
/** Имя GIF файла, содержащего изображение персонажа.
* Подразумевается, что на этом изображении персонаж стоит лицом налево.
* Выбрать рост персонажа относительно роста других персонажей, добавляя
* сверху и снизу прозрачные пикселы. Самый высокий персонаж не должен
* иметь этих добавочных пикселов.
*/
private String imageFileNameI = null;

/*Конструкторы*/

/** Равномерно распределить исходное число очков-жизней по всем характеристикам.
* <p> Требование: 3.2.ПВ.1.2 (инициализация значений характеристик)
*/
protected EncounterCharacter()
{   super();
    for( int i = 0; i < qalityTypeS.length; ++i )
        qualValueI[i] = QUAL_TOTAL_INIT / qualityTypeS.length;
}

/** Создать новый персонаж, используя заданное имя и файл с изображением.
* <p> Требование: 3.2.ПВ.1.1 (присвоение имени персонажу)
* @param  nameP  Печатаемое имя персонажа
* @param  imageFileP  Имя файла относительно файла с изображением персонажа
*/
protected EncounterCharacter( String nameP, String imageFileP )
{   this();
    setName( nameP);
    imageFileNameI = imageFileP;
}

/** Создание нового персонажа с использованием заданного имени.
* <p> Требование: 3.2.ПВ.1.1 (присвоение имени персонажу)
* @param  nameP  Печатаемое имя персонажа
*/
```

продолжение

Листинг 7.6 (*продолжение*)

```
protected EncounterCharacter( String nameP)
{   this( nameP, null );
}

/*Методы*/

/** Требование 3.2.ПВ.3.2 (возможность настраивать значения
* характеристик персонажа игры)
* Благодаря синхронизации значение qualityValueI постоянно даже при наличии
* других потоков.
* <p> SDD-ссылка: 6.1.2.1.1
* <p> Инварианты: см. инварианты класса
* <p> Предусловия: qualityP принадлежит qualityTypeS[]
*   AND qualityValueP >= 0
*   AND qualityValueP <= сумма значений характеристик
* <p> Постусловия: qualityP имеет значение qualityValueP
*   AND значения остальных характеристик остаются в том же соотношении, что и
    раньше, за исключением значений менее некоторого допустимого - они
    приравниваются к нулю.
* @param  qualityP  Характеристика, значение которой мы будем настраивать
* @param  qualityValueP  Значение, которое мы присвоим этой характеристике
*/
public synchronized void adjustQuality(String qualiryP, float qualityValueP)
{   // Значение характеристики, которое следует изменить
    float qualityValueM = qualityValueI[indexOf( qualityP )];

    // Сохраняем сумму значений
    float originalSumM = sumOfQualities();

    // pc Присвоить заданное значение, учитывая пороговую величину
    setQuality( qualityP, qualityValueP );
    // pc Если настраивается единственная ненулевая характеристика,
    // распределить введенное значение поровну между
    // остальными характеристиками
    if( originalSumM == qualityValueM )
    {  float qualityDiffEach=(originalSumM-
qualityValueP)/(qualityTypeS.length-1);
        for( int i = 0; i < qualityTypeS.length; ++i )
            if( !qualityTypeS[i].equalsIgnoreCase( qualityP ) )
                setQuality( qualityTypeS[i], qualityDiffEach );
    }
    else {
        /* Множитель (proportionM), на который следует изменить все
        * остальные характеристики.
        * Пример: пусть были значения 1, 3, 5 (то есть сумма равна 9),
        * и первую характеристику изменили с 1 на 2. Тогда для «3» и «5»
        * сумма должна измениться с 8/9 от общей на 7/9, так что каждое из
        * этих чисел следует умножить на 7/8, то есть на (9-2)/(9-1).
        */
```

```
        float proportionM=(originalSumM-qualityValueP)/(originalSumM-qualityValueM);

        //pc Настроить остальные характеристики, сохраняя их соотношение
        for( int i = 0; i < qualityTypeS.length; ++i )
            if( !qualityTypeS[i].equalsIgnoreCase( qualityP ) )
            setQuality( qualityTypeS[i], qualValueI[i] * proportionM);
        }
}

/** Получить копию списка имен значений характеристик.
* @return   Рабочие копии строк имен, представляющие характеристики.
*/
public static String[] getQualityTypes()
{
    // Скопировать массив строк
    String [] returnListM = new String[qualityTypeS.length];

    // Скопировать каждую строку
    for( int i = 0; i < qualityTypeS.length; i++ )
        returnListM[i] = new String(qualityTypeS[i]);

    // Вернуть копию
    return returnListM;
}

/**  Возвращает значение определенной характеристики.
* <p> Предусловие: qualityP принадлежит qualityTypeS[]
* @param  qualityP  Характеристика, значение которой мы хотим получить
* @return           Значение определенной характеристики
*/
public float getQualityValue( String qualityP )
{
    return qualValueI[ indexOf( qualityP ) ];
}

/** Значения характеристик ниже этого порогового значения обнуляются
* во избежание зависания программы на неопределенное время.
* <p> Требование: например, 3.2.ПВ.1.2 (нижняя граница ненулевых значений
* характеристик)
* @return   Допустимое значение
*/

static final float getTolerance()
{   return 0.5f;
}

/** Возвращает индекс заданной характеристики.
* <p> Предусловие   qualityP принадлежит qualityTypeS[]
```

продолжение

Листинг 7.6 (*продолжение*)

```
* @param   qualityP   Характеристика, которую мы ищем
* @return             Индекс характеристики
*/
private static int indexOf( String qualityP )
    {
        // По умолчанию для ненайденных значений
        int returnIndexM = -1;

        // Поиск в таблице имен характеристик
        for( int i = 0; i < qualityTypeS.length; ++i )
        if( qualityTypeS[ i ].equalsIgnoreCase( qualityP ) ) //Нашли имя?
            {   returnIndexM = i;        // Запомнить значение индекса
                break;
            }

        return returnIndexM;
}

/** Установить по умолчанию максимальное число букв в именах персонажей.
* <p> Требование:   3.2.ПВ.1.1 (ограничение на длину имени персонажа)
* @return   Максимальное число символов, разрешенное в имени персонажа
*/
protected int maxNumCharsInName()
{   return 15;
}

/** Установить значение характеристики без учета значений других
* характеристик. Отбросить значения ниже минимального порога, обнулив
* значение характеристики.
* Синхронизация не допускает изменений в qualityValueI, даже если она
* используется и другими потоками.
* <p>Требования:  3.2.ПВ.2 (нижний предел ненулевых значений характеристик)
* <p>Предусловие:  qualityP принадлежит qualityTypeS[]
* <p>Постусловие:  Значения характеристик выше минимально допустимого или
*                  равны нулю.
*
* @param    qualityP   Характеристика, которой присваивается новое значение.
* @param    valueP     Новое значение, присваиваемое характеристике.
*/
public synchronized void setQuality( String qualityP, float valueP )
{
    if( valueP < getTolerance() )
        qualValueI[ indexOf( qualityP ) ] = 0.0f;
    else
        qualValueI[ indexOf( qualityP ) ] = valueP;
}
```

```
/** Изобразить персонаж
* <p> Требования: 2.1.2.1 (персонаж, изображенный в зоне игры)
*   SRS 3.2.ПИ.1 (выбор изображения персонажа)
*   SRS 3.2.ХИ.1 (изображение персонажа в окне установки характеристик)
* @param   compP       Компонент пользовательского интерфейса, в котором
*                      персонаж должен быть нарисован
* @param   drawP       Графический контекст для рисования
* @param   posP        Пиксельные координаты в compP для центра изображения
* @param   heightPixP  Желаемая высота изображения, в пикселах
* @param   faceLeftP   <tt>true</tt> Если персонаж повернут лицом налево,
*                      <tt>false</tt> если персонаж повернут лицом направо
*/
public void showCharacter( Component compP, Graphics drawP, Point posP,
    int heightPixP, boolean faceLeftP )
{
    if( imageFileNameI == null )
    {   // Не указано имя файла. Вывести вместо него имя персонажа.
        drawP.setColor( Color.magenta );
        FontMetrics fm = drawP.getFontMetrics();
        // Вывести имя по центру расположения персонажа
        drawP.drawString( getName(),
            posP.x - fm.stringWidth( getName() ) / 2,
            posP.y - fm.getHeight()/2 );
    }

    else {  // Имя файла предоставлено. Показать изображение из файла
        Image chImageI = compP.getToolkit().getImage( imageFileNameI );

        // Размеры изображения
        int imageWidth = chImageI.getWidth(compP);
        int imageHeight = chImageI.getHeight(compP);
        // Отмасштабировать ширину и высоту
        int scaledWidth = imageWidth * heightPixP / imageHeight;

            // Считайте, что обычно персонаж повернут лицом налево.
            // Решите, развернуть изображение или нет.
        if( faceLeftP )
        // Нарисовать изображение как есть, отмасштабировать и отцентровать
            drawP.drawImage( chImageI,
                posP.x - scaledWidth/2, posP.y - heightPixP/2,
                posP.x + scaledWidth/2, posP.y + heightPixP/2,
                0, 0, imageWidth-1, imageHeight-1, null );
        else
        // Нарисовать изображение, повернув персонаж лицом в другую сторону,
        // отмасштабировать и отцентровать
            drawP.drawImage( chImageI,
                posP.x + scaledWidth/2, posP.y - heightPixP/2,
```

продолжение

Листинг 7.6 (*продолжение*)

```
                posP.x - scaledWidth/2, posP.y + heightPixP/2,
                0, 0, imageWidth-1, imageHeight-1, null );
        }
}

/** Вычисляет сумму значений характеристик.
* Благодаря синхронизации гарантируется, что другой поток не изменит
* qualityValueI в тот момент, когда этот поток вычисляет сумму

* <p> Требования:  3.2.ПВ.3.2 (пропорции значений характеристик)
* @return          Сумма значений характеристик игрока, >= 0
*/

public synchronized float sumOfQualities()
{
    float sumM = 0.0f;

    for( int i = 0; i < qualityTypeS.length; ++i )
        sumM += qualValueI[i];
    return sumM;
}

} //Конец класса EncounterCharacter
```

Глава 8
Модульное тестирование

...в этих людях,
Отмеченных хотя б одним изъяном,
...
Все их достоинства — пусть нет им счета
И пусть они, как совершенство, чисты, —
По мненью прочих, этим недостатком
Уже погублены.

Шекспир. Гамлет

В связи с потенциальным ущербом от каждой программной ошибки и возрастающей сложностью нахождения и исправления ошибок по ходу роста программы раннее и частое тестирование становится важной частью процесса разработки.

Сразу после реализации частей программы мы немедленно приступаем к тестированию (рис. 8.1).

- Разделы 8.1–8.6
- Упражнения
- Пример: Индивидуальная программная документация (PSD), часть 2

Вопросы, рассматриваемые в этой главе:

- Понимание значения модульного тестирования.
- Различие между тестированием методами «черного» и «белого ящика».
- Разработка адекватной зоны охвата.
- Изучение стандартов тестирования.
- Инспектирование плана модульного тестирования.

8.1. Введение в модульное тестирование

8.1.1. Цели тестирования

Мы не можем протестировать программу абсолютно во всех аспектах, поскольку число вариантов работы нетривиальной компьютерной программы может быть неограниченным. Следовательно, тестирование не может доказать отсутствия ошибок в программе, в то время как доказательство корректности способно это сделать. Тестирование может только показать *присутствие* ошибок.

Рис. 8.1. План разработки программ: темы главы 8

Тестирование часто неправильно воспринимается как процесс подтверждения корректности кода, что можно выразить таким высказыванием: «Протестируй это, чтобы убедиться, что тут все правильно». Иногда, однако, это является целью тестирования, особенно незадолго до отсылки продукта или при регрессионном тестировании (объясняется в следующей главе). Главная цель тестирования далека от подтверждения корректности. Его цель не в том, чтобы показать удовлетворительную работу программы, а в том, чтобы четко определить, в чем работа программы неудовлетворительна!

Время, использованное на тестирование, требует значительных затрат, и мы стараемся получить от этих затрат максимальную прибыль. Для данной тестируемой программы, чем больше дефектов будет найдено на каждый доллар зарплаты, тем выше выигрыш от вложений в тестирование. Следовательно, целью тестирования является обнаружение как можно большего числа дефектов с высоким уровнем важности. Резюмируя сказанное выше, перечислим «золотые правила» тестирования.

- Цель тестирования: максимизировать число и важность обнаруженных дефектов на каждый затраченный доллар.
- Поэтому: начинайте тестирование рано.
- Ограниченные возможности тестирования: тестирование может установить только присутствие дефектов, и никогда — их отсутствие.
- Для установления факта *отсутствия дефектов* используйте доказательства корректности.

Тестирование оценивается более чем половиной времени, затраченного на проект. Наградой за нахождение дефекта на ранней стадии процесса является по крайней мере десятикратная экономия по сравнению с обнаружением этого же

дефекта на этапе интеграции или, еще хуже, после отправки заказчику. Следовательно, мы должны тестировать *рано* и *часто*.

Что касается идеальной гарантии качества в общем, тестирование кода должны проводить люди, не участвовавшие в его разработке. Когда инженер разрабатывает код, он создает для себя представление того, что код должен выполнять. Поэтому в то же время он разрабатывает типичную среду, в которой этот код должен выполняться. Можно смело считать, что код дает немного ошибок в этой конкретной среде. Следовательно, эта среда является основой тестов разработчика. Именно поэтому, когда человек тестирует свой собственный код, он часто прячет каждый дефект, который необходимо найти.

Модульное тестирование является ранним типом тестирования. Следующий уровень состоит из *интегрального тестирования*. Здесь валидируется общая функциональность каждой стадии конкретной программы. Наконец, система и различные приемосдаточные тесты валидируют финальный продукт, как описано в следующей главе. Уже разработанные варианты использования также берутся в качестве основы для некоторых из этих тестов. Типы тестирования и связь между ними проиллюстрированы на рис. 8.2.

Рис. 8.2. Тестирование: общая картина

В этой главе рассказывается о модульном тестировании; все другие варианты тестирования описаны в главе 9.

8.1.2. Значение модульного тестирования

Цель модульного тестирования — проверить структуру, в то время как цель всех других видов тестирования обычно заключается в проверке функциональности. В качестве аналогии представьте себе тестирование каждой опоры моста на заводе. Это является неким подобием модульного тестирования, поскольку в этом случае тест затрагивает элементы структуры. Тест, состоящий из проезда автомобиля по частично сконструированному мосту, напротив, не будет модульным тестированием. Функции обычно являются наименьшими частями программы,

к которым может быть применено модульное тестирование (см. рис. 8.2). Следующим по величине элементом является модуль (класс в случае объектно-ориентированной ориентации). Иногда комбинации модулей рассматриваются в целях тестирования как модули.

Модули, к которым применяется модульное тестирование, являются блоками при построении программы, а не отдельными кирпичами, на которых строится дом. Хотя на дом не сильно повлияют несколько бракованных кирпичей, программное приложение может оказаться очень чувствительным к дефектам в отдельных блоках конструкции. Если дефектные части будут встроены в программы, может понадобиться огромное количество времени на их нахождение и исправление. Поэтому блоки программы должны быть абсолютно надежными, что и является целью модульного тестирования.

Модульное тестирование является дополнением к инспектированию и использованию формальных методов проверки корректности.

В терминологии USDP модульное тестирование проводится преимущественно на итерациях проектирования, а также на ранних этапах итерации конструирования [64] (рис. 8.3).

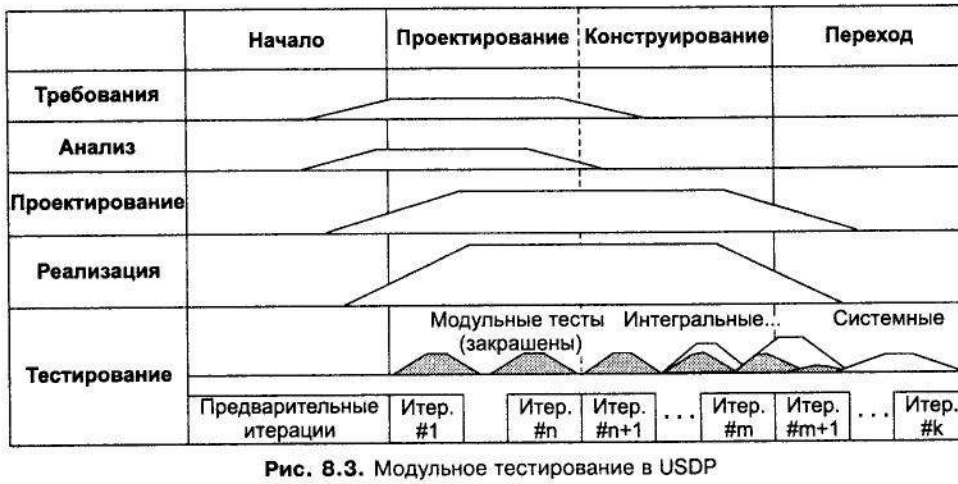


Рис. 8.3. Модульное тестирование в USDP

8.1.3. Типичный план модульного тестирования

Типичный план модульного тестирования, основанный на стандарте IEEE 1008-1987, показан на рис. 8.4. Далее объясняются шаги процесса модульного тестирования.

1. Входными данными для процесса планирования теста являются требования и детальный проект. Вспомните, что каждое требование соответствует хорошо определенному коду (функции, где возможно). Детальный проект обычно состоит из дополнительных классов и методов. Они также сказываются на качестве программы и должны быть протестированы в том же объеме, что и отдельные требования. Выходными данными процесса плани-

рования теста является модульный план тестирования (например, «(1) тест метода 84; (2) тест метода 14; ...; (*m*) тест класса 26, ...»).

2. Следующим шагом является получение входных и выходных данных, ассоциирующихся с каждым тестом. Некоторые из этих данных могут быть уже получены из предыдущих тестов (например, предыдущие итерации, предыдущие версии продукта или предыдущие выпуски). Результатом на этом шаге является набор тестов.
3. Наконец, тесты исполняются.

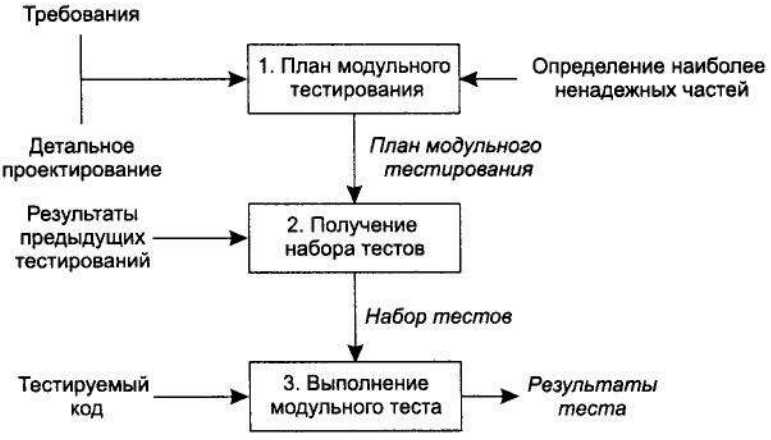


Рис. 8.4. План модульного тестирования

Далее приведены этапы, требуемые стандартом IEEE для модульного тестирования (1008-1987). Они расширяют только что описанный план модульного тестирования.

1. Спланировать общий подход, ресурсы и расписание.
2. Определить характеристики, которые следует протестировать, исходя из требований.
3. Обновить общий план.
4. Разработать набор тестов.
5. Реализовать обновленный план и проект.
6. Выполнить тестовые процедуры.
7. Проверить окончание работы.
8. Оценить тестирование и модули.

8.2. Типы тестов

Хотя Гленфорд Майерс написал свою книгу «The Art of Software Testing» [82] еще в 1978 году, эта небольшая книга остается классикой, и она сыграла свою роль в написании этой главы.

8.2.1. Тестирование «черного ящика», «белого ящика» и «серого ящика»

В этом разделе будет дано определение тестирования «черного», «белого» и «серого ящика». В остальной части главы будет описано, как планировать, проектировать и выполнять такие тесты.

Когда мы интересуемся исключительно тем, как программа или ее часть предоставляет соответствующие выходные данные, мы тестируем ее на каждое требование, используя подходящие входные данные. Это называется тестированием «черного ящика», поскольку мы не обращаем внимания на то, что находится внутри «ящика» (программы): «ящик» может быть «черным». Тесты «черного ящика» могут быть эффективны, если мы можем убедиться, что они исчерпывают все комбинации входных данных. Это докажет заказчику, что все требования удовлетворены. Однако никакое тестирование не охватывает всех возможностей.

Тестирование «черного ящика» похоже на тестирование моста путем проезда по нему нескольких комбинаций различных транспортных средств. Это неэффективно, поскольку нам нужно проверить и составные части моста, и то, как они объединены в систему. Последнее называется идеей «тестирования белого ящика». Тестирование «черного ящика» и «белого ящика» проиллюстрировано на рис. 8.5.

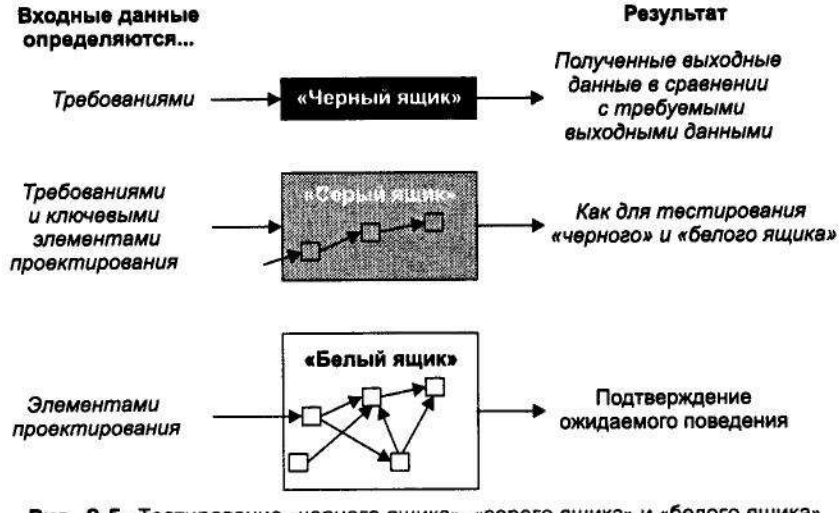


Рис. 8.5. Тестирование «черного ящика», «серого ящика» и «белого ящика»

Целью тестирования «белого ящика» является тестирование наиболее ненадежных путей программы. Для выполнения тестирования «белого ящика» мы сначала разбиваем проект программы на отдельные элементы и ищем пути и другие разбиения для управления и данных. Затем мы проектируем тесты, прослеживающие все или некоторые из этих путей, и проверяем все составные части. Более наглядным названием этих действий было бы «тестирование стеклянного ящика».

Тестирование «серого ящика» рассматривает внутреннюю работу программы или модуля, но только до некоторой степени. Сюда могут быть также отнесены и некоторые аспекты тестирования «черного ящика».

8.2.2. Разбиение равнозначности для тестирования «черного ящика»

Поскольку у нас нет возможности протестировать все комбинации входных данных, мы ищем представительные варианты тестов. Набор возможных вариантов тестов для трех переменных в финансовой программе — капитал, процентная ставка и оценка инфляции — изображен на рис. 8.6. Проблема заключается в нахождении наилучшего представления бесконечного множества возможностей наиболее представительным определенным множеством. *Разбиение равнозначности* — это разбиение входных тестовых данных на подмножества таким образом, чтобы при условии успешного выполнения одного из них остальные элементы подмножества также с большой вероятностью прошли бы тест успешно. Например, при тестировании функции `setName( String )` в классе `GameCharacter` успешное завершение тестового вызова `setName( «Harry» )` означает, что у нас, вероятно, не будет проблем при тестировании функции `setName()` с любой строкой из пяти символов. Более того, мы, вероятно, можем расширить это разбиение равнозначности на «все имена не менее чем с одним и не более чем с `maxNumCharsInName()` символами».

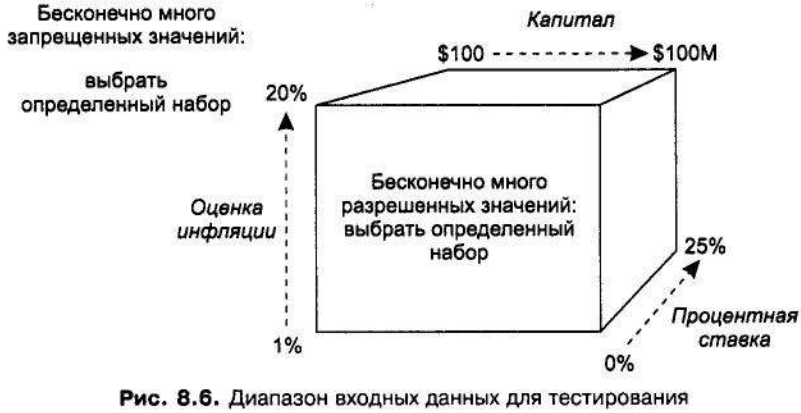


Рис. 8.6. Диапазон входных данных для тестирования

Разбиение равнозначности для метода, вычисляющего величину в зависимости от заданного капитала, процентной ставки и оценки инфляции, представлено на рис. 8.7. Например, область, закрашенная на рис. 8.7 серым, соответствует следующему разбиению равнозначности:

«оценка инфляции между 15 и 20 %», «капитал от $65 до $100» и «процентная ставка между 0 и 5 %».

Максимальная прибыль от тестирования обычно достигается при анализе граничных значений, о котором речь пойдет далее.

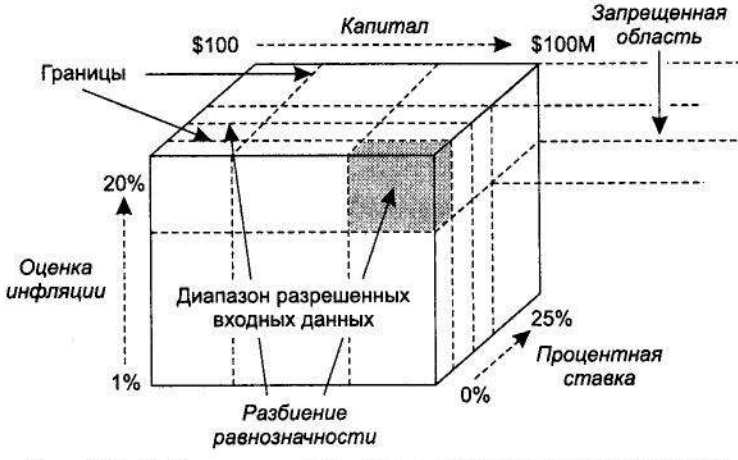


Рис. 8.7. Разбиение равнозачности входных данных тестирования

8.2.3. Анализ граничных значений для тестирования «черного ящика»

К разбиениям равнозначности обычно прибегают при исследовании граничных значений внутренних переменных программы. Например, оценка инфляции должна лежать между 1 и 20 %, что дает две границы. Предположим, что значения инфляции до 15 % и значения, превышающие эту величину, программа обрабатывает по-разному. Это дает нам еще одну дополнительную границу (рис. 8.7).

В функции `setName( String )`, упоминавшейся ранее, границами считаются строки нулевой длины и строки длиннее `maxNumCharsInName()`. Границы могут быть также представлены в форме уравнения, например $x + y = 7$. Оно может быть выражено условием типа `while(x+y<=7)`.

В тестировании проекта значения, лежащие за пределами этих границ (например, недействительный ввод данных), также используются в качестве тестовых данных. Тестовые данные генерируются после того, как будут установлены границы эквивалентных классов (рис. 8.8).

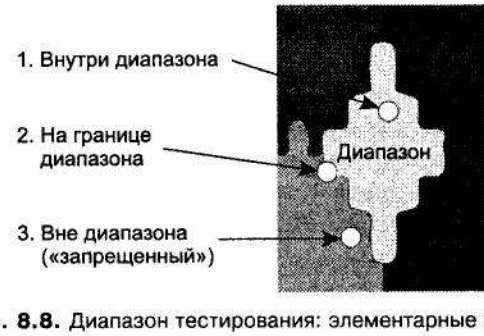


Рис. 8.8. Диапазон тестирования: элементарные случаи

8.2.4. Рассмотрение утверждений для тестирования «белого ящика»

Каждое утверждение в программе должно быть выполнено по крайней мере одним тестом. Анализ каждого утверждения *обязателен*. В качестве примера (рис. 8.9) покажем, что выполнение всех утверждений не гарантирует корректности программы (см. [82]). Требования для программы на рис. 8.9 определены блок-схемой. Тестовый вариант u==2, v==0, x==3 выполняет каждую команду реализации и генерирует корректные выходные данные (x==2,5). Однако программа дефектна, так как в ней не реализована блок-схема. Например, в случае u==3, v==0, x==3 программа генерирует ответ x==1, хотя должно быть x==2, как показано на блок-схеме.

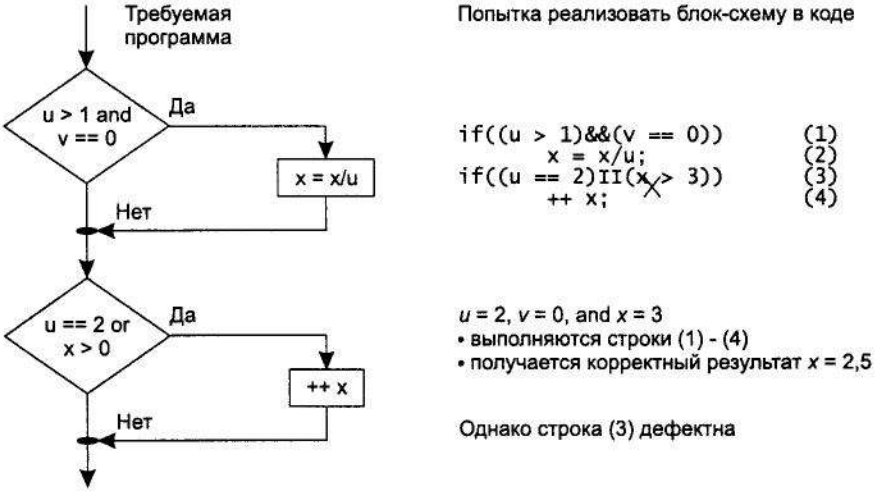


Рис. 8.9. Область охвата теста «Каждое утверждение не является достаточным»

8.2.5. Рассмотрение решений для тестирования «белого ящика»

Обзор решений гарантирует, что программа принимает каждую ветвь каждого решения. Например, рассмотрим блок-схему на рис. 8.10. Убедимся, что каждое *да* и *нет* можно получить в результате хотя бы одного теста из набора тестов.

Циклы эффективно реализуют последовательность условных операторов. Например, цикл

```
for( I = 0; I < 3; ++i)
        v[i] = w[I+1] + w[i];
```

можно «раскрыть» в такую последовательность условных операторов:

```
// For i==0:
i = 0;
    v[i] = w[i+1] + w[i];
```

```
    ++i;
// For i==1:
if (i < 3){
    v[i] = w[i+1] + w[i];
    ++i;
}
// For i==2:
if (i < 3){
    v[i] = w[i+1] + w[i];
    ++i;
}
```

Рис. 8.10. Направления, которые необходимо проверить

Обзор решений гарантирует выполнение каждой итерации каждого цикла. Его можно реализовать путем определения каждого цикла и подбора входных данных таким образом, чтобы каждая итерация каждого цикла выполнилась хотя бы один раз. Этот подход не так сложен для циклов `for`, однако он представляет некоторые сложности для циклов `while`. Например, как бы мы стали тестировать итерацию следующего цикла:

```
while(u < v) {...} // ?
```

Иногда все возможные варианты можно просчитать, иногда их можно разбить на типовые группы. Однако в некоторых случаях полное рассмотрение решений с помощью циклов `while` практически невозможно. Вспомните, что циклы `while` часто допускают применение формальных методов и инспектирования. Это

иллюстрирует вспомогательную природу формальных методов, проверок и тестирования.

Рассмотрение решений обычно содержит в себе рассмотрение утверждений, поскольку все последующие точки ветвления во всех комбинациях обычно включают в себя каждое утверждение кода. Рассмотрение решений может оказаться недостаточным из-за того, что в некоторых решениях могут скрываться другие. Например, в утверждении

```
if( A && B )…
```

условие B никогда не тестируется, если условие A ложно. Аналогично, в утверждении

```
if( A || B )…
```

условие B никогда не тестируется, если условие A истинно.

Для решения таких проблем можно использовать *тестирование с множественными условиями*. Это тестирование представляет собой полный объем условного тестирования, проверяющий каждую комбинацию каждого условия по крайней мере один раз. Разработка таких тестовых вариантов может оказаться довольно скучным занятием, поскольку необходимо проследить в программе каждое условие, чтобы определить подходящие входные данные. Важную роль в создании этих тестовых комбинаций играет программное обеспечение, генерирующее тесты автоматически.

До сих пор мы уделяли основное внимание проверке того, что все условия выполняются и что результаты получаются ожидаемыми. Эта идея лежит в основе техники «серого ящика», согласно которой мы тестируем входные и выходные данные («черный ящик») наряду со всеми утверждениями («белый ящик»). Нам также нужно проверить, что по ходу работы программа проходит через все предполагаемые состояния. Эту задачу решает тестирование инвариантов.

8.2.6. Тестирование на основе инвариантов

Вспомните, что *инварианты* — это утверждения, связывающие переменные. Они выражают состояния. Например, для выражения состояния банкомата после того, как клиент вставил карту и ввел корректный личный код, используется нижеследующий типичный инвариант:

```
( cardInserted==true ) && ( pin == VALID )
```

Как уже отмечалось в главе 7, мы используем инварианты для поддержки интеллектуального управления вычислениями и для доказательства корректности кода. Во многих случаях эти утверждения остаются неизменными в стратегически важных блоках кода. Часто бывает полезно добавлять в исходный код команды, сообщающие о том, действительно ли инвариант, который мы предполагаем истинным, сохраняется таковым во время работы программы. Эта техника «белого ящика» называется *тестированием, основанным на инвариантах*. Например, если сумма переменных x и y во время вычисления должна быть равной 10, во время работы программы можно проверять инвариант `assert(x + y == 10);`.

Функция assert(<аргумент>) может сообщить об истинности или ложности своего аргумента. Желательно, чтобы вычисления приостанавливались в том случае, когда инвариант ложен, и т. д. Иногда функции assert() оставляют в исходном коде и для работы программы, например:

```
assert( potentiallyFatalXRayDose < 137 );
```

хотя обычно инварианты используются только для проверки корректности программы.

Предположим, что мы хотим использовать тестирование инвариантов для программы *max()* из раздела 7.4 о доказуемой корректности программ. Инварианты, которые должны быть удовлетворены, можно выразить способом, показанным в листинге 8.1.

Листинг 8.1. Тестирование на основе инвариантов

```
public static boolean checkAssertion
    ( int loopSoFarP, int indexOfMaxP, double[] arrayP )
{
    // Определение блока проверки
    // Сначала установите следующие булевы переменные:

    boolean b1M = true; /* Означает, что значения arrayP[] меньше
                arrayP[indexOfMaxP] для всех индексов <indexOfMaxP> */
    if( indexOfMaxP != 0 )
        for ( int u = 0; u < indexOfMaxP; ++u )
            b1M &= ( arrayP[u] < arrayP[indexOfMaxP] );
    boolean b2M = true; /*Означает, что значения arrayP[] не более
            arrayP[indexOfMaxP] для индексов indexOfMaxP ... loopSoFarP*/
    for( int v = indexOfMaxP; v <= loopSoFarP; ++v )
        b2M &= ( arrayP[v] <= arrayP[indexOfMaxP] );
    if
    (               // Цикл дошел до индекса loopSoFarP
                    ( 0 <= loopSoFarP ) && ( loopSoFarP < arrayP.length )
                    && //IndexOfMaxP - это индекс <= loopSoFarP...
                    ( 0 <= indexOfMaxP ) && ( indexOfMaxP <= loopSoFarP )
                    && b1M && b2M // ...где встретили первый максимум
    )
    {               System.out.println( "Утверждение верно" );
                    return true;
    }
    else
    {               System.out.println( "Утверждение неверно" );
                    retuen false;
    }
}
```

Теперь можно применять тестирование инвариантов каждый раз, когда предполагается, что инвариант будет истинным в программе (листинг 8.2). Пример с программой *max()* является очень простым, но количество проблем, связанных

с тестированием инвариантов, несоизмеримо с размером тестируемого кода. Однако для более сложного кода, подлежащего тестированию, время настройки может иногда оказаться разумным, а выгода (количество ошибок, найденных за каждый затраченный час) гораздо выше.

Листинг 8.2. Применение тестирования на основе инвариантов к max()

```
/** Находит индекс и значение первого из наибольших элементов массива */
public static void main( String [] mainArg )
{
double a[] = getArray();

//Пусть I - инвариант ... (раздел 4 главы 7). Устанавливаем I.
int i=0;
int k=0;
boolean validityM = checkAssertion( i, k, a );  //Проверка инварианта.
                                                //Применение тестирования
                                                //инвариантов к max()

//Следующий код поддерживает I истинным (раздел 4 главы 7).
while( i1=a.length-1)
{   ++i;
        if( a[i]>a[k] )
        k = i;
validityM = validityM && checkAssertion(i, k, a );  //Проверка инварианта.
                                                //Применение тестирования
                                                //инвариантов к max()
}

//Сообщение о результатах тестирования
System.out.println( "Первый наибольший элемент массива равен" + a[k] + "и имеет
номер" + k );
System.out.println( "Иваринат имеет значение:" + validityM );
}
```

Современные языки программирования либо имеют встроенные возможности тестирования инвариантов (C++ и Eiffel), либо позволяют построение функций тестирования инвариантов программному инженеру или третьей стороне.

8.2.6.1. Ограничения проверки автоматических инвариантов

У проверки инвариантов есть свои ограничения. Например, инварианты, использованные в примере НОД в разделе 7.4, невозможно проверить в существующей форме с помощью блока проверки утверждений. Вспомните, что инвариантное утверждение, использовавшееся для доказательства, имело вид:

$$НОД(anX, aY) = НОД(x, y)$$

С точки зрения выполнения, это затрагивает основное вычисление (наибольшего общего делителя), которое мы пытаемся реализовать! С другой стороны,

люди *могут* использовать свое понимание НОД для убеждения друг друга (и самих себя) в корректности кода.

8.2.7. Использование случайных величин в тестировании

Вообще говоря, мы используем случайные входные данные при тестировании с целью избежать возникновения необъективных экземпляров тестов. Например, если мы хотим оценить мнение населения о политическом деятеле, мы выбираем представителей населения случайным образом. Здесь случайный выбор применяется определенным образом: сначала определяется тип теста, а затем применяется случайная выборка для получения объективных тестовых данных. То же справедливо и для программного тестирования. Когда выбран тип теста (например, рассмотрение решений) и определены границы данных, для возможных входных данных существует некоторая свобода. В идеале эти входные данные должны выбираться случайным образом.

В качестве примера представьте себе выполнение рассмотрения решений для блок-схемы из раздела 8.2.4. Предположим, мы хотим, чтобы ответом на вопрос «Имя слишком длинное?» было «Да». Существует большое множество данных, для которых это будет справедливо, и именно из этого множества и следует в идеале случайным образом выбирать тестовый пример. Один из способов — выбрать каждое имя, как указано далее.

Выбрать целочисленное i больше, чем `maxNumCharsInName()`, случайным образом.

Для $j = 0...i$ выбрать случайным образом букву и установить в `name[j]` эту букву.

В качестве второго примера представьте выполнение тестирования граничных значений для программы, проверяющей, являются ли четыре вещественных числа из последовательности $x1$, $x2$, $x3$ и $x4$ допустимыми значениями для эксперимента. Допустим, что допустимыми будут значения, удовлетворяющие неравенствам:

$-5 < x1 \leq 10$
$x1 + x2 \leq x3$
$x3 \geq x4$

Для граничного тестирования нам необходимо выбрать тестовые данные со следующими ограничениями:

$x1 = -5$ (недопустимо); $x1 = 10$
$x1 + x2 = x3$
$x3 = x4$

Обратите внимание, что переменная $x1$ должна иметь только одно из двух значений, $x3$ определена сразу после того, как выбраны $x1$ и $x2$, а $x4$ определяется через $x3$. Поэтому мы должны выбрать из бесконечного множества значений $x2$, что мы и делаем случайным образом во избежание предвзятости.

8.3. Планирование модульных тестов

Систематический подход в тестировании необходим, поскольку число потенциальных модулей, нуждающихся в тестировании, обычно очень велико. Достаточно легко сказать, что «каждая часть работы должна быть протестирована», однако эта фраза несет в себе мало смысла, поскольку на этап тестирования выделяется лишь ограниченное количество ресурсов. Поэтому цель заключается в нахождении как можно большего количества ошибок как можно более серьезного уровня в рамках имеющихся ресурсов.

ОДИН ИЗ СПОСОБОВ СПЛАНИРОВАТЬ МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

1. Определите принципы модульного тестирования.
   - Назначить самого разработчика ответственным?
   - Предоставить проверку другим?
   - Предоставить проектирование и проверку другим?
2. Определите, что, где и как документировать.
   - Индивидуальная документация?
   - Как и где внедрять в другие типы тестирования?
   - Внедрить ли в формальные документы?
   - Использовать ли инструменты и тестовые утилиты?
3. Определите объемы модульного тестирования заранее:
   - не тестируйте просто «пока время не кончится»;
   - расставляйте приоритеты, чтобы важные тесты точно были проведены.
4. Определите, как и где получить тестовые входные данные.
5. Оцените необходимые ресурсы.
   - Используйте имеющиеся данные предыдущих проектов, если это возможно.
6. Организуйте учет времени, подсчет ошибок, фиксацию их типа и источника.

1. Определите принципы модульного тестирования. Первый вопрос заключается в определении того, какие модули мы будем рассматривать и кто будет их тестировать.

Для объектно-ориентированных проектов обычная организация модульного тестирования заключается в тестировании методов каждого класса, затем классов каждого пакета, затем пакета в целом. В нашем примере мы будем сначала тестировать классы в каждом пакете, а затем перейдем к программным пакетам, поскольку тестирование последних зависит от тестирования первых.

Тестирование модуля в идеале планируется и выполняется человеком, не участвовавшим в разработке. Модульное тестирование в реальной жизни иногда планируется и исполняется организацией контроля качества. Хотя достоинством такого подхода является независимость тестирования, в этом случае от инженеров организации контроля качества требуется понимание проекта в деталях. Некоторые организации-разработчики не поддерживают эту возможность и поэтому требуют от контролирующей качество организации только тестирования высокого уровня. Модульное тестирование часто остается за группой разработчиков и выполняется по их собственному усмотрению. В любом случае тесты делаются доступными для инспектирования и для возможного внедрения более высокого

уровня. В некотором смысле независимость организации контроля качества может быть заменена перекрестным тестированием, выполняемым самими разработчиками, когда они тестируют модули друг друга.

2. Определить способ документирования модульных тестов. Документирование модульных тестов состоит из тестовых процедур, входных данных, кода, исполняющего тест, и выходных данных. Модульные тесты можно упаковывать вместе с кодом либо в отдельных документах. Достоинство упаковки тестов вместе с кодом заключается в удобстве. Эта техника проиллюстрирована в разделе 8.4.2.1. Недостатком является увеличение объема кода. Можно использовать предварительную компиляцию для урезания тестового кода перед компиляцией модуля.

Кроме того, для выполнения модульных тестов используются драйверы тестов и утилиты. Они также документируются для будущего использования. Например, в нашем учебном проекте используется следующая служебная тестовая функция:

```
public static void reportToFile
    (
    FileWriter FileWriterP,
    String TestDescriptionP,
    String CorrectOutputStringP,
    String ActualOutputStringP
    )
```

3. Определите объемы модульного тестирования. Поскольку «протестировать все» невозможно, границы тестирования должны быть сознательно определены. Например, если банковская программа состоит из *снятия*, *депозитов* и *запросов*, модульное тестирование могло бы указать, что каждый метод должен быть протестирован с равным количеством разрешенных, граничных и запрещенных данных или, возможно, методы *снятия* и *депозита* должны тестироваться в три раза больше, чем методы *запросов*, и т. д. В общем случае методы, изменяющие состояние (значения переменных), обычно тестируются больше других. Границы того, что относится к модульному тестированию, также должны быть определены. Например, входит ли сюда тестирование пакетов, или оно должно относиться к другому типу тестирования (глава 9)?

Разработчики заранее определяют границы тестирования, в том числе и момент, когда процесс тестирования должен быть завершен. Например, следует ли тестировать каждый модуль одинаковое количество времени или до обнаружения первых трех ошибок? Критерии остановки тестирования обсуждаются далее в разделе 8.5.2.

Вспомните, что наша идея тестирования заключается в выполении тестов, которые с наибольшей вероятностью помогут выявить ошибки. Расставляя приоритеты тестам в соответствии с вероятностью обнаружения ими ошибок, мы тем самым стараемся оптимизировать время, отведенное на тестирование. Приоритетный подход зависит от тестируемого модуля. Основным источником ошибок являются циклы. Сюда же относятся граничные значения и интерфейсы. В последней категории тесты, затрагивающие части модулей, такие как функции, обычно изобилуют потенциальными ошибками, поскольку каждая часть ожидает получить данные в определенной форме, а эти ожидания часто не выполняются.

4. Определите, как и где получать тестовые входные данные. Мы обсудили разрешенные, граничные и запрещенные входные тестовые данные. Также необходима некоторая случайная генерация данных. По возможности используются инструменты, генерирующие входные тестовые данные посредством анализа исходного кода и обнаружения граничных значений данных и ветвления. Вдобавок значительный объем тестовых данных можно получить из предыдущих версий программы, стандартных источников, промышленных контрольных задач и т. д. Все это документируется для будущих ссылок и повторного использования.

5. Оцените необходимые ресурсы. Как всегда при планировании, мы определяем человеко-месяцы и время, необходимое для выполнения модульного тестирования. Основным источником этой оценки являются накопленные данные. Хотя модульное тестирование часто связано с процессом разработки, отдельное его выполнение предоставляет бесценную информацию.

6. Организуйте учет времени, а также учет дефектов, их типа и источника. Участвующие инженеры определяют четкую форму, в которой они будут вести учет затраченного на модульное тестирование времени, учет ошибок и их типов. Полученные данные используются для утверждения состояния программы и предсказания конечного качества работы и сроков окончания. Данные также становятся частью учетных записей истории проекта.

Следующие два раздела представляют примеры планирования модульных тестов на уровнях методов и классов. Пример в конце главы показывает получившийся код.

8.4. Контрольные таблицы и примеры тестирования методов

8.4.1. Модульное тестирование методов

Хэмфри [48] советует использовать для выполнения тестирования методов контрольные таблицы.

ОДИН ИЗ СПОСОБОВ ВЫПОЛНИТЬ ТЕСТИРОВАНИЕ МЕТОДОВ ПО ХЭМФРИ (1) —

1. Проверить работу при нормальных значениях параметров.
   Тест «черного ящика», основанный на требованиях к модулю.
2. Проверить работу при граничных значениях параметров.
   «Черный ящик».
3. Проверить работу при значениях параметров вне разрешенного диапазона.
4. Убедиться, что выполняются все инструкции.
   Рассмотрение утверждений.
5. Проверить все пути, в том числе обе ветви каждого узла.
   Рассмотрение решений.
6. Проверить использование всех вызванных объектов.
7. Проверить обработку всех структур данных.
8. Проверить обработку всех файлов.

ОДИН ИЗ СПОСОБОВ ВЫПОЛНИТЬ ТЕСТИРОВАНИЕ МЕТОДОВ ПО ХЭМФРИ (2) —

9. Проверить нормальное завершение всех циклов.
   (Часть доказательства корректности.)
10. Проверить непредусмотренное завершение всех циклов.
11. Проверить нормальное завершение всех рекурсий.
12. Проверить непредусмотренное завершение всех рекурсий.
13. Проверить обработку всех условий ошибок.
14. Проверить синхронизацию и расчет времени.
15. Проверить аппаратные зависимости.

Методы принадлежат одной из двух категорий (рис. 8.11). Первая категория соответствует требованиям, предъявленным к программе. Вторая категория работает с классами и методами, добавленными для формирования проекта.

Рис. 8.11. Связь тестов с требованиями и проектированием

Модульное тестирование включает в себя автономное тестирование по возможности каждого метода на точное соответствие требованию, сформулированному в SRS. Другими словами, мы проверяем, что метод удовлетворяет своему требованию. В данных условиях это представляет собой тестирование «черного ящика». Также используется и тестирование «белого ящика» применительно к каждому методу (например, для рассмотрения утверждений и решений).

Для методов, возникающих из проекта, мы часто не располагаем явно сформулированными требованиями, в соответствии с которыми можно было бы выполнить тесты. Примером теста, происходящего из проекта, является тест класса *СостояниеИгры*. Это класс, введенный для проектирования нашей видеоигры: ни одно из исходных требований не касается конкретно этого класса. В идеале для всех классов проекта при создании таковых должны быть написаны отдельные требования. Когда такие отдельные требования не написаны, как часто случается, тестовые варианты следует разрабатывать в соответствии с предполагаемой функциональностью класса.

8.4.2. Пример теста метода

План для выполнения тестирования модуля метода в случае проекта *Встреча* может быть таким.

Для каждого класса разрабатывается метод, тестирующий методы. Он выполняет каждый метод класса с различными значениями параметров. В методе тестирования используется вспомогательный метод reportToFile() класса TestExecution.

Метод reportToFile() имеет следующие параметры:

<файл назначения> <выходная строка с комментариями> <полученное значение> <ожидаемое значение>.

Для получения тестовых данных по каждому методу мы вручную определяем границы параметра, затем вручную выбираем хотя бы одно значение внутри границ, на границах и за разрешенными границами.

Пример, приведенный ниже, показывает, как выбираются значения для одного метода. Код в конце главы демонстрирует применение плана тестирования для класса *ПерсонажВстречи*. Как будет видно далее, разработка систематических тестовых вариантов даже для этих случаев не так проста.

Мы будем тестировать следующее D-требование видеоигры *Встреча*:

3.2.ПВ.1.2. У каждого игрового персонажа имеется одинаковый набор характеристик. Каждая характеристика представлена неотрицательным числом с десятичной запятой и как минимум одной цифрой в дробной части. Все они одинаково инициализируются, так что сумма значений всех характеристик равна 100. Значение характеристики не может быть в промежутке 0–0,5.

В первой версии будут такие характеристики, как сосредоточенность, ум, терпение, выносливость и сила.

Подходящий набор тестов для метода `adjustQuality( String qualityP, float valueP )` приводится ниже. Этот метод устанавливает значение характеристики `qualityP` равным `valueP` и настраивает остальные характеристики так, чтобы их взаимные соотношения оставались прежними. Чисто случайная комбинация разрешенных и неразрешенных значений `qualityP` и `valueP` *не* будет подходящим началом разработки набора тестов: во-первых, нужно обдумать диапазон параметров. В каждом диапазоне нужно определить граничные, внешние и лежащие внутри диапазона данные. Мы стремимся использовать системный подход, что обычно означает поиск представителей разбиения равнозначности. Типичный результат систематического разбиения входных данных на зоны эквивалентности показан в табл. 8.1.

Таблица 8.1. Разбиение значений на диапазоны для модульного тестирования

Модульный тест			Применили к	
			сосредоточенности	выносливости
		Возвращает нуль?		
1. Внутри диапазона		Нет: Тест 1.1	Тест 1.1.1	
			...	...
		Да: Тест 1.2	Тест 1.2.1	
				...
2. На границах	2.1. Вызвали с нулевым значением	Нет: Тест 2.1.1	Тест 2.1.1.1	
				...
		Да: Тест 2.1.2	...	
	2.2. Вызвали со значением 100	...		
	...			
3. Вне диапазона	3.1. Вызвали с параметром < 0			
	...			

Второй уровень разбиения можно определить исходя из того, может ли значение характеристики оказаться нулевым в результате применения метода `adjustQuality()`. Третий уровень разбиения показан в табл. 8.1, а результаты перечислены ниже.

1. Тестовые варианты с параметрами из диапазона

1.1. adjustQuality() возвращает ненулевое значение:

1.1.1. Параметр характеристики = «сосредоточенность»

Входные данные (в идеале следует выбрать здесь число от 0 до 100, чтобы в сумме получалось число менее 100):

Сосредоточенность: 20;

Выносливость: 20 (замечание: ¼ всех значений без сосредоточенности);

Ум: 20;

Терпение: 20;

Сила: 20.

Выполнение: `adjustQuality( «сосредоточенность», 10)` (В идеале это значение выбирается случайным из внутреннего диапазона, гарантированно дающего ненулевое значение сосредоточенности).

Ожидаемый результат:

Сосредоточенность: 20 + 10 = 30;

Выносливость: 70 / 4 = 17,5 (замечание: ¼ всех значений без сосредоточенности);

Ум: 70 / 4 = 17,5;

Терпение: 70 / 4 = 17,5;

Сила: 70 / 4 = 17,5.

Тесты 1.1.2, 1.1.3, ... аналогичны, используют другие характеристики вместо сосредоточенности.

1.2. adjustQuality() возвращает нулевое значение:

(В идеале использовать случайную величину из диапазона аналогично описанному выше способу.)

1.2.1. Параметр характеристики = «сосредоточенность»

Входные данные:

Сосредоточенность: 20;

Выносливость: 20 (замечание: ¼ всех значений без сосредоточенности)

Ум: 20;

Терпение: 20;

Сила: 20.

Выполнение: **adjustQuality(«сосредоточенность», 99)**

Ожидаемый результат:

Сосредоточенность: 99;

Выносливость: 0 (¼ заменена на нуль);

Ум: 0 (? заменена на нуль);

Терпение: 0 (¼ заменена на нуль);

Сила: 0 (¼ заменена на нуль).

Тесты 1.2.2, 1.2.3, ... аналогичны, используют другие характеристики вместо сосредоточенности.

2. Тестовые варианты с параметрами на границах диапазона

2.1. adjustQuality() вызван с нулевым значением одного параметра:

2.1.1. Результаты при ненулевых значениях:

2.1.1.1. Параметр характеристики = «сосредоточенность»

Входные данные:

Сосредоточенность: 0;

Выносливость: 25

Ум: 25;

Терпение: 25;

Сила: 25.

Выполнение: **adjustQuality(«выносливость», 74)**

Ожидаемый результат:

Сосредоточенность: 0;

Выносливость: 99;

Ум: 0 (1/3 заменена на нуль);

Терпение: 0 (1/3 заменена на нуль);

Сила: 0 (1/3 заменена на нуль).

Тесты 2.1.1.2, 2.1.1.3, ... аналогичны

Тесты 2.2, 2.3, ... соответствуют другим границам. Например:

2.2.N. adjustQuality() вызван со значением, равным текущему:

Входные данные:

Сосредоточенность: 0;

Выносливость: 25

Ум: 25;

Терпение: 25;

Сила: 25.

Выполнение: **adjustQuality(«выносливость», –25)**

Ожидаемый результат:

Сосредоточенность: 0;

Выносливость: 0;

Ум: 33;

Терпение: 33;

Сила: 33.

3. Тестовые варианты с параметрами вне диапазона

3.1. Выше верхней границы значений параметров

3.1.1. Параметр характеристики = «сосредоточенность»

Входные данные:

Сосредоточенность: 20;

Выносливость: 20;

Ум: 20;

Терпение: 20;

Сила: 20.

Выполнение: **adjustQuality(«сосредоточенность», 81)**

Ожидаемый результат:

Сообщение об ошибке, докладывающее о том, что метод `adjustQuality()` был вызван с неразрешенным параметром.

Сосредоточенность: 100 (20 + 81 установили на 100);

Выносливость: 0 (после установки сосредоточенности не осталось очков-жизней для распределения между остальными характеристиками);

Ум: 0;

Терпение: 0;

Сила: 0.

Тесты 3.1.2, 3.1.3, ... аналогичны

Тесты 3.2, 3.3, ... аналогичны тесту 3.1

3.N. Ниже нижней границы значений параметров

3.N.1 Параметр характеристики = «сосредоточенность»

Входные данные:

Сосредоточенность: 20;

Выносливость: 20;

Ум: 20;

Терпение: 20;

Сила: 20.

Выполнение: **adjustQuality(«сосредоточенность», –21)**

Ожидаемый результат:

Сообщение об ошибке, докладывающее о том, что метод `adjustQuality()` был вызван с неразрешенным параметром.

Сосредоточенность: 0 (20 – 21 установили на 0);

Выносливость: 25 (100/4);

Ум: 25 (100/4);

Терпение: 25 (100/4);

Сила: 25 (100/4).

Остальные тесты создаются аналогично.

8.4.2.1. Тестирование всех методов класса

В качестве примера рассмотрим класс `GameCharacter` (*ПерсонажИгры*) пакета `Characters` (*Персонажи*). На данный момент это единственный класс данного пакета, и он уже рассматривался на этапе создания архитектуры.

Мы покажем код внутри класса `GameCharacter`, который будет выполнять тестирование метода `setName()`. Это можно организовать так, как показано далее. (Обозначение `//ps` указывает на фрагмент для автоматического выделения псевдокода.) В примере в конце главы представлен полный тест метода.

```
/*
* GameCharacter.java
*/
package FrameworkRPG.Characters;
```

```
import TestUtilities.*; //внутренний тестовый пакет
import java.util.*;
import java.io.*;

/**
* Персонаж ролевой игры.
* @author Eric Braude
* @version     0.1, 7/14/98
*/

public abstract class GameCharacter
{
// АТРИБУТЫ - - - - - - - - - - - - - - - - - - - - -
/** Имя игрового персонажа; при инициализации null */
private String nameI;

// МЕТОДЫ - - - - - - - - - - - - - - - - - - - - - -
/** Ни одно имя не будет длиннее этой величины */
public final static int limitOfNameLength()
{
return 100;
}

/** Для регистрации */
protected void display()
{
System.out.println( «Game character » + nameI + « displayed» );
}

/** Доступ к nameI. «defaultName» назначается при первом
*     доступе
*/
public String getName()
{
if( nameI == null ) //раньше не было доступа
        setName( «defaultName» );
return nameI;
}

/** Подклассы должны объявлять ограничение на размер имен
*     персонажей
*/
protected abstract int maxNumCharsInName();

/** Устанавливает значение nameI равным nameP, если длина
*   не превосходит maxNumCharsInName(); в противном случае
```

```
*   обрезается.
*   Наследники должны использовать это для setName( String ),
*   но не перезаписывать.
*   @param nameP: предлагаемое имя для nameI
*/
protected final void setName( String nameP )
{
//ps IF параметр строки OR maxNumCharsInName() не имеет смысла
if( ( nameP == null ) || ( maxNumCharsInName() <= 0 ) || ( maxNumCharsInName() >
limitOfNameLength() ) )
{
// Установить в качестве имени значение по умолчанию
// и показать его в системном окне
        nameI = new String( «defaultName» );
        System.out.println
        («defaultName selected by GameCharacter.setName()»);
}
// ps ELSE
else
        //ps IF строковый параметр слишком длинный
        if( nameP.length() > maxNumCharsInName() )
            //ps обрезать по maxNumCharsInName()
            nameI = new String ( nameP.getBytes(),
                                 0, maxNumCharsInName() );
        else
        //ps ELSE назначить имя параметру
            nameI = new String( nameP );
}

/** Протестировать этот класс */
public static void main( String[] args )
{
        //Получить имя файла для результатов теста и
        //запустить тест
        String outputFileNameM;
        if( args == null || args.length == 0 )
        {
            System.out.println( «Using output.txt as default
            for    rest results." );
            OutputFileNameM = «output.txt»;
        }
        else
            outputFileNameM = args[0];
        try
        {
            testGAmeCharacterMethods( outputFileNameM );
        }
```

```
        catch( IOException e )
        {
            System.out.println( e ):
        }
}

/** Протестировать все методы этого класса */
public static void testGameCharacterMethods(
            String destinationP ) throws IOException
{
        //подготовка к тесту:
        FileWriter outM = new FileWriter(
                        new File( destionationP ) ):

// поскольку это абстрактный класс, нужно создать один
// конкретный для тестирования
        class TempGameCharacter extends GameCharacter
        {
            public int maxNumCharsInName()
            {
                return 20;
            }
        }
        TempGameCharacter c = new TempGameCharacter();

        System.out.println
        ( «Test for Gamecharacter class with
            maxNumCharsInName() =" + c. maxNumCharsInName() );

        System.out.println( «GameCharacter test results on» +
            DestinationP + «\n» );
}

// Тесты для SetName() - - - - - - - - - - - - - - - - - - - -
/** SetName() 1. Разбиение равнозначности: номинальное,
* разрешенное имя.
*/
c.setName( «Harry» );
TestExecution.reportToFile( outM, «SetName Test 1.1:
        Non-null string", c.nameI, «Harry» );

/* SetName() 2. Анализ граничных значений */
c.setName( «X» ); //минимальная разрешенная длина
TestExecution.reportToFile( outM, «SetName Test 2.1:
        Min non-null string", c.nameI, «X» );
```

Планы модульного тестирования для `getName()`, `display()` и `getCharacter()` выполняются аналогично. В примере в конце главы показан тест для методов класса *ПерсонажВстречи*.

8.5. Контрольные таблицы и примеры тестирования классов

После тестирования отдельных методов класса мы можем продолжать тестировать класс в целом. Тестирование класса представляет собой совместное выполнение методов класса или тестирование объектов класса при определенных событиях, например событиях мыши. Абсолютно случайный набор комбинаций методов и в этом случае, скорее всего, приведет к пустой трате времени и не закроет пробелы в рассмотрении класса. Однако существует несколько дополнительных способов тестирования классов.

ОДИН ИЗ СПОСОБОВ ВЫПОЛНИТЬ МОДУЛЬНОЕ ТЕСТИРОВАНИЕ КЛАССОВ

1. Испытывайте комбинации методов:
   - обычно из 2–5 методов;
   - выберите сначала наиболее общие последовательности;
   - учтите последовательности, которые могут привести к ошибке;
   - результирующие значения атрибутов требуется подсчитывать вручную.
2. Фокусируйте модульные тесты на каждом атрибуте.
   - Инициализируйте атрибут, а затем запускайте последовательности методов, влияющих на него.
3. Проверяйте, что инвариант каждого класса не изменяется:
   - проверьте, что инвариант истинен при начальных значениях;
   - выполните последовательность (например, ту же, что и на шаге 1);
   - проверьте, что после этого инвариант остается истинным.
4. Проверьте, что объекты переходят из ожидаемых состояний в ожидаемые:
   - спланируйте последовательность переходов состояний;
   - установите объект в исходное состояние, присвоив значения переменным;
   - обеспечьте выполнение первого события и проверьте выполнение перехода, и т. д.

План выполнения тестирования на уровне класса для примера видеоигры *Встреча* может быть таким:

Для каждого класса X разрабатывается метод testXClass(), выполняющий методы в различной последовательности. testXClass() можно выполнять в методе main() или во внешнем для этого класса коде. Последовательности методов разрабатываются посредством техник, изложенных на рис. №№.

8.5.1. Пример теста комбинации методов

Каждый тест комбинации методов состоит из последовательности вызовов функций. Например, в классе *ПерсонажВстречи* следующие методы будут тестироваться последовательно:

Сокращение	Прототип метода
aq	adjustQuality(String qualityP, float qualityValueP)
d	deleteFromEncounterCharacters(EncounterCharacter EncounterCharacterP)
ge	EncounterCharacter getEncounterCharacter(String nameP)
gq	float getQualityValue(String qualityP)
gs	float getSumOfQualities()
gt	float getTolerance()
io	int indexOf(String qualityP) throws Exception
ii	InsertIntoEncounterCharacters(EncounterCharacter encounterCharacterP)
m	int maxNumCharsInName()
sq	setQuality(String qualityP, float valueP)

Мы концентрируем наши тестовые ресурсы двумя способами. Мы определяем:

- последовательности, которые, вероятно, будут чаще всего использоваться;
- последовательности, которые наиболее подвержены возникновению ошибок. Это приводит к нахождению наибольшего числа ошибок на каждый затраченный доллар.

Приведенные ниже последовательности часто встречаются:

- ge-aq-gs // получить персонаж — настроить характеристики — получить сумму характеристик;
- ge-sq-aq-gq // получить персонаж — установить значение характеристики — настроить характеристики — получить характеристику.

Вас не должно пугать бесконечное количество возможных тестов, которые можно запустить даже для несложных программ. (Ввиду этого инспектирование и доказательство корректности не так дороги, как они могут показаться на первый взгляд.) На длину последовательностей методов накладывается ограничение: поскольку у нас десять методов, мы ограничим максимальную длину числом 10. Подход отбраковки может оказаться полезным для определения общих последовательностей. Данную последовательность методов можно классифицировать как наиболее или наименее подверженную возникновению ошибок. В противном случае последовательность относят к категории «ни то, ни другое». Все «наиболее вероятные» последовательности тестируются. Также тестируется как можно большее число последовательностей из категории «ни то, ни другое». Последовательности из «наименее вероятных» тестируются, если на это есть время.

Поскольку процесс настройки значений характеристик относительно сложен, ниже приведен пример последовательности, которая с большой вероятностью может порождать ошибки:

- ge-sq-aq-sq-aq-gq /* получить персонаж — установить значение характеристики — настроить характеристики — установить значение характеристики — настроить характеристики — получить характеристику */

8.5.2. Тесты атрибутов

Тесты атрибутов разрабатываются посредством фокусировки на отдельных атрибутах и предсказания воздействия на них различных методов из последовательности, например `setBalance(100); addToBalance(70); getBalance()`. Мы выполняем последовательность и проверяем, чтобы получившееся значение атрибута оказывалось таким, какое мы ожидали. В результате тесты атрибутов представляют собой тесты последовательностей методов с проверкой атрибутов.

8.5.3. Тестирование инвариантов класса

Как рассказывалось в главе 7, инварианты класса являются ограничениями на атрибуты класса, которые должны сохраняться истинными в соответствующих точках выполнения. Тесты инвариантов классов состоят из проверки истинности каждого инварианта посредством выполнения последовательности методов и проверки сохранения истинности инварианта. Например, один из инвариантов класса *ПерсонажВстречи* заключается в том, что сумма значений характеристик должна быть менее 100. Ниже приведен фрагмент кода, который проверяет этот инвариант. Он взят из метода `testEncounterCharacterClass` класса *ПерсонажВстречи*.

```
/*. . . . . .
Проверить инвариант «сумма значений характеристик объекта персонажа chapacterP <= 100"
*/
boolean truthM = true;
float sumM = 0.0;
for( int i = 0; i < qualityTypeS.length; ++i )
{
        // накапливаем сумму значений характеристик
        sumM += characterP.qualValueI[i];
        // установить равным нулю, если sumM > 100
        truthM = truthM && ( sumM <= 100.0 );
}
// сравнение ожидаемого и полученного результатов
TestExecution.printReportToFile
(
        outM, // файл журнала тестирования
        «Проверить инвариант «сумма значений характеристик объекта персонажа
        chapacterP <= 100'», //описание теста
        truthM, // полученное значение
        true    // ожидаемое значение
);
```

Некоторые языки (например, Eiffel) и некоторые интерфейсы прикладного программирования (например, SunTest) снабжены функциями, тестирующими истинность инвариантов, которые обычно называются `assert(…)`.

8.5.4. Тесты на основе состояний

Как мы уже видели, объекты классов часто можно представить как переход между состояниями в ответ на события. Такие классы мы должны протестировать на основе их состояний. Например, протестируем класс *ИграВстреча*. Первые шаги в тестировании состояний программы показаны на рис. 8.12. Полный тест переходов состояний показан на рис. 8.13. Числа показывают типичный порядок событий, приводящий к имеющей смысл последовательности событий. Один тест поэтому должен представлять собой подачу таких входных данных, чтобы система проходила эту последовательность состояний. Также следует рассмотреть события, которые *не* применимы ни к одному конкретному состоянию, чтобы убедиться, что они не влияют на ход работы программы.

Рис. 8.12. Тестовая последовательность переходов состояний в видеоигре Встреча

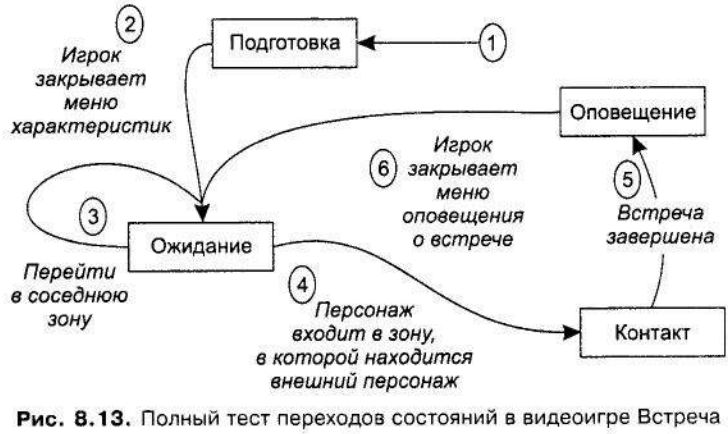


Рис. 8.13. Полный тест переходов состояний в видеоигре Встреча

На планирование и выполнение тестов на основе состояний необходимо потратить значительное время. Это в первую очередь связано с необходимостью в исчерпывающих входных данных. Существуют инструменты, записывающие взаимодей-

ствие пользователя с системой, которые могут затем воспроизвести эти действия. Помимо этого можно использовать проверку инвариантов в коде, чтобы удостовериться в том, что система находится в том состоянии, в каком должна.

8.6. Подведение итогов

Модульное тестирование состоит из тестирования структурных компонентов находящейся в разработке программы. Этим они отличаются от комплексных тестов, проверяющими функциональные части программы по ходу разработки.

Тестирование «черного ящика» проверяет, чтобы тестируемый код предоставлял корректные выходные данные по всем входным данным. Тестирование «белого ящика» проверяет процесс получения выходных данных. У тестирования «белого ящика» существует несколько аспектов. Тестировать следует *рано и часто*. Разработанные заранее планы тестов помогают пролить свет на требования. Подведем итоги этой главы.

- Модульное тестирование — тестирование «частей».
- Интегральное тестирование — тестирование «всего в целом».
- Метод «черного ящика» проверяет обработку:
  - выполняемую многими способами;
  - позволяет убедиться в законченности.
- Планирование теста: чем раньше, тем лучше.
- Помогает прояснить требования.

Упражнения

Ответы и подсказки для упражнений, помеченных символами «о» или «п», приводятся в конце этой главы.

Вопросы для проверки

П8.1°. Опишите уровни тестирования 3–4.

П8.2°. Опишите различия между тестированием «черного», «белого» и «серого ящика». Ответ на вопрос находится в разделе 8.2.

П8.3°. Протестировать все комбинации входных данных невозможно. Назовите технику «черного ящика», задачей которой является уменьшение числа тестовых вариантов.

П8.4°. Тестируем выборочно.

1. Сформулируйте практический критерий выбора входных тестовых данных.
2. Вообще говоря, какие виды входных значений обычно приводят к большинству общих ошибок?

П8.5°. Приведите пример программы со следующими свойствами. Программа должна реализовывать простую блок-схему, но осуществляет это некорректно. Существуют входные данные к программе, которая выполняет каждую строку программы и возвращает корректный ответ. Раздел 8.2.4 показывает решение этого вопроса.

П8.6°. Приведите пример программы в форме блок-схемы с единственной точкой ветвления. Создайте дефектную программу со следующим свойством. Существуют входные данные, исполняющие каждую ветвь программы. Ответ на этот вопрос содержится в разделе 8.2.5.

П8.7°. Какое утверждение вы бы стали проверять при компиляции, находящей первый минимум в массиве *a* размером 100?

П8.8°. Назовите 4–6 шагов в планировании модульного тестирования. Ответом на этот вопрос является раздел 8.3.

П8.9°. Назовите 3 теста «черного ящика» для функций (методов).

П8.10°. Назовите 6–12 тестов «белого ящика» для функций (методов).

П8.11°. Приведите код, проверяющий, удовлетворяется ли инвариант *I* в классе *ПерсонажВстречи*. *I* = «все значения характеристик неотрицательны». Обратитесь к переменным в фрагменте кода *ПерсонажВстречи* в разделе 8.5.3.

П8.12°. Опишите тестирование на основе состояний. Ответ на этот вопрос вы найдете в разделе 8.5.4.

Упражнения в команде

К8.1. («Модульное тестирование») Выполните полное модульное тестирование двух основных методов вашей программы. Опишите, сколько времени члены вашей команды отдельно и все вместе потратили на разработку каждой части этих тестов и как этот процесс можно было бы улучшить.

Критерии оценки.

1. Степень четкости формулировки плана («Отлично» — очень четкий план).
2. Степень, в которой в план и тест были включены все существенные аспекты модульного тестирования («Отлично» — все важные рассмотрения, упомянутые в этой главе).
3. Реализм самооценки и план улучшения («Отлично» — очень конкретные данные по результатам и конкретные изменения для улучшения процесса).

Общие упражнения

О8.1. Напишите код для класса *Счет* с атрибутом *_баланс*, методами доступа и методом *добавить()*. Исходите из того, что *Счет* имеет состояния *Платежеспособный*, *Пустой* и *Задолженность*, и они реализованы с использованием образца проектирования State. Напишите полный набор модульных тестов для класса *Счет*, в том числе и тесты на основе состояний.

Ответы

П8.1. Функциональные тесты проверяют каждую отдельную функцию или метод. Тесты класса проверяют каждый класс. Затем идут тесты пакетов. Комплексные тесты проверяют части программы. Наконец, системные тесты проверяют программу в целом.

П8.3. Разбиение равнозначности уменьшает число вариантов тестов посредством разбиения множества возможных входных данных на отдельные подмножества. Эти подмножества выбираются так, чтобы при удачном прохождении теста с одним набором данных все возможные входные данные из этого подмножества тоже скорее всего были удачны в тестировании.

П8.4.

1. Проверить входные данные, которые наиболее вероятно дадут ошибку.
2. Дефекты обычно появляются чаще всего в граничных случаях.

П8.7. Утверждение, которое нужно удовлетворить, является следующим набором из четырех утверждений.

0 <= i < *a.length – 1* // прогресс цикла до индекса *i*
AND
0 <= k <= i // минимумом на этот момент является элемент с индексом *k*
AND // первый минимум – *k*-элемент
[либо k = 0
либо *[a[u] > a[k] для u = 0, 1, ..., k–1]* //значения больше у всех элементов с индексом менее *k*
AND
[a[u] <= a[k] для u =k, k+1, ..., i] // между элементами *k* и *i* элементов нет меньших чисел

П8.9. Тесты функций «черного ящика» можно получить с помощью следующих тестовых данных:

- из диапазона;
- на границе диапазона;
- вне диапазона.

П8.10.

1. Проверить все пути, в том числе обе ветви каждого ветвления.
2. Убедиться, что все инструкции были выполнены.
3. Проверить использование всех вызванных объектов.
4. Проверить обработку всех структур данных.
5. Проверить обработку всех файлов.
6. Проверить правильное завершение всех циклов.
7. Проверить неправильное завершение всех циклов.
8. Проверить правильное завершение всех рекурсий.
9. Проверить неправильное завершение всех рекурсий.
10. Проверить обработку всех условий ошибок.

11. Проверить время и синхронизацию.
12. Проверить все аппаратные зависимости.

П8.11. Приведенный ниже код проверяет инвариант класса, согласно которому все значения характеристик должны быть неотрицательными.

```
// . . .
// Проверить инвариант e.equalValue[i] >= 0
boolean truthM = true;
for( int i = 0; i < qualityTypeS.length; ++i )
        //Установить truthM в значение false, если любое
        //вхождение в e.qualValue[i] не >= 0
        truthM = truthM && (e.qualValueI[i] >= 0);
// сравнение ожидаемого и полученного результатов
TestExecution.printReportToFile
(
        out, // test log file
        «Проверить инвариант 'e.equalValue[i] >= 0'», //описание теста
        truthM, // полученное значение
        true    // ожидаемое значение
);
```

Пример. Индивидуальная программная документация (PSD) на Java для класса EncounterCharacter (ПерсонажВстречи), часть 2

[Примечание для студентов. Ниже приведена вторая часть документа, описывающего индивидуальную программную документацию для EncounterCharacter (ПерсонажВстречи). Формат этого документа взят из IEEE-стандарта для документации по тестированию программы.

То, как выполняется модульное тестирование в этом примере, является лишь одним из многочисленных способов. Например, альтернативным путем было бы выполнение тестов через статические самотестирующие методы из внешнего объекта. Этот объект можно сделать так, чтобы он выполнял несколько модульных тестов и посылал результаты в конкретные выходные файлы, следуя инструкциям в тестовом скриптовом файле.]

4. Модульное тестирование для класса EncounterCharacter (ПерсонажВстречи)

4.1. Спецификация проекта тестирования

Модульное тестирование для EncounterCharacter (ПерсонажВстречи) состоит из двух открытых методов:

- `testEncounterCharacterMethods()` проверяет каждый метод по очереди.
- `testEncounterCharacterClass()` проверяет последовательности методов.

Эти методы можно выполнить из метода `EncounterCharacter main()` с помощью внешнего объекта.

4.2. Спецификация совокупности тестовых данных

В `testEncounterCharacterMethods()` и `testEncounterCharacterClass()` встраиваются совокупности тестовых данных для `EncounterCharacter`.

[Примечание для студентов. Для простоты этот модульный тест содержит данные внутри метода. Однако, обычно входные данные и ожидаемый результат извлекаются из файла.]

4.3. Спецификация процедуры тестирования

Модульные тесты для `EncounterCharacter` инициируются посредством выполнения метода `main()`. Параметр, передающийся в `main()`, определяет файл, в который записываются результаты.

[Примечание для студентов. Это простая процедура, однако она значительно усложняется, когда требуется взаимодействие исходных файлов и пользователя. Например, это будет в модульном тестировании класса EncounterGame (ИграВстреча).]

4.4. Документация результатов тестирования

Документация результатов тестов состоит из журнала испытаний, отчета о тестах и итогового тестового отчета.

4.4.1. Журнал испытаний

[Примечание для студентов. Учет результатов теста. См. пример ниже.]

Содержится в файле: EncounterCharacter_Test_Log_day_month_year.doc.

4.4.2. Отчет о тестах

[Примечание для студентов. Любые события, требующие внимания во время тестов. См. пример ниже.]

Содержится в файле: EncounterCharacter_Test_Incident_day_month_year.doc.

4.4.3. Итоговый тестовый отчет

Содержится в файле: EncounterCharacter_Test_Summary_Report__day_month_year.doc.

Пример журнала испытаний (PSD, раздел 4.4.1)

EncounterCharacter_Test_Log_26_Jul_1999

Протестированные методы:

>>>>>GetCharacter тест 1: номинальное значение <<<<<
querty< — — получено
querty< — — требовалось получить
>>>>>GetCharacter тест 2: значения внешнего параметра<<<<<
defaultName< — — получено
defaultName < — — требовалось получить

```
>>>>>EncounterCharacter тест 3: граничные значения параметра<<<<<
123456789012345< — — получено
123456789012345< — — требовалось получить
Ожидается одно имя каждого персонажа
querty
defaultName
123456789012345
>>>>>indexOf() тест 1: правильное название характеристики<<<<<
фактическое целое число = ожидаемому целому числу
>>>>>indexOf() тест 2: правильное название характеристики<<<<<
фактическое целое число = ожидаемому целому числу
>>>>>setQuality() тест 1: номинальное значение<<<<<
фактическое вещественное число = ожидаемому вещественному числу
>>>>>setQuality() тест 2: номинальное значение<<<<<
фактическое вещественное число = ожидаемому вещественному числу
>>>>>adjustQuality() тест 0: проверить то, что сумма значений равна 100<<<<<
фактическое вещественное число = ожидаемому вещественному числу
>>>>>adjustQuality() тест 1: проверить то, что сумма значений равна 100 после изменения<<<<<
фактическое вещественное число = ожидаемому вещественному числу
>>>>>adjustQuality() тест 2: проверить то, что значения изменились так, как нужно<<<<<
фактическое вещественное число = ожидаемому вещественному числу
>>>>>adjustQuality() тест 3: проверить то, что маленькие значения обнуляются<<<<<
фактическое вещественное число = ожидаемому вещественному числу
>>>>>adjustQuality() тест 4: проверить то, что сумма значений равна 100 после настройки<<<<<
фактическое вещественное число = ожидаемому вещественному числу
Тест класса:
>>>>>Классовый тест ge-aq-so<<<<<
100,0< — — получено
100,0< — — требовалось получить
>>>>>Классовый тест ge-aq-aq-gq-so: часть 1<<<<<
20,9876< — — получено
20,9876< — — требовалось получить
>>>>>Классовый тест ge-aq-aq-gq-so: часть 2<<<<<
100,0< — — получено
100,0< — — требовалось получить
>>>>>Классовый тест инварианта '_qualValue[i]>=0'<<<<<
true< — — получено
true< — — требовалось получить
```

[Примечание для студентов. Пример журнала тестирования не показывает неудачные тесты. Их можно описать здесь, вынести в отдельный файл.]

Пример отчета о результатах теста (PSD, раздел 4.4.2) EncounterCharacter_Test_Incident_26_Jul_1999.doc

Тестирование проводилось для версии 7.2.1 EncounterCharacter с использованием пакета TestUtilities версии 2.3. С первой попытки мы не смогли запустить тест. Мы считаем, что это было связано с тем фактом, что у нас на самом деле не было версии 2.3 TestUtilities. Когда мы перезагрузили этот пакет, тест прошел без проблем.

[Примечание для студентов. Здесь уместно упомянуть об ошибках, сделанных во время тестирования. Это уместно особенно в тех случаях, когда требуются действия пользователя, а производить перезапуск всего теста непрактично.]

Пример итогового тестового отчета (PSD, раздел 4.4.3)

EncounterCharacter_Test_Summary_26_Jul_1999.doc

Этот тест проводился Джоном Джонсоном в 14.00 с использованием версии 1.1.6 виртуальной машины Sun. Тест прошли 100 % методов модульного теста. Эти методы были добавлены Э. Брауде в версию 6.5.2. Они будут дорабатываться в дальнейших версиях EncounterCharacter.

Пример исходного кода модульного теста

Приведенный в листинге 8.3 код для класса EncounterCharacter содержит методы, тестирующие сами себя.

Класс TestExecution используется для выполнения модульного тестирования. Он содержит статический метод printReportToFile(), методы которого в нотации Javadoc приведены ниже.

```
*@param-FileWriter-местонахождение файла с отчетом
*@param-String-описание теста
*@param-int-ожидаемый корректный результат
*@param-int-полученный результат
* @return-void
*@exception-нет
```

Здесь нет предусловий. Постусловием является требование того, чтобы файл был записан по соответствующему адресу, указанному во входном параметре FileWriter. Он должен содержать описание тестовых входных данных, ожидаемый результат и полученный результат — каждый четко указан. Код показан в листинге 8.3.

Листинг 8.3. Исходный код для модульного тестирования класса EncounterCharacter (ПерсонажВстречи)

```
/** Тестирование данного класса
* @param  argsP   Расположение журнала тестирования методов относительно
*                 журнала тестирования классов
*/
public static void main( String[] argsP )
{
        // Файлы, в которых по умолчанию записываются выходные
        // данные тестов и фиксируется работа тестов
        String methodOutputFileNameM = «methodOutput.txt»;
```

продолжение

Листинг 8.3 (*продолжение*)

```
        String classOutputFileNameM= «classOutput.txt»;

        // Если входные данные некорректны, использовать
        // значения по умолчанию
        if( argsP != null && argsP.length == 2 )
        {   methodOutputFileNameM = argsP[0];
            classOutputFileNameM  = argsP[1];
        }

        // 1. ВЫПОЛНИТЬ ТЕСТЫ, НЕ ТРЕБУЮЩИЕ УЧАСТИЯ ЛЮДЕЙ

        // Протестировать методы индивидуально, затем протестировать классы
        try
        {   testEncounterCharacterMethods( methodOutputFileNameM );
            testEncounterCharacterClass( classOutputFileNameM );
        } catch( IOException eP )
        {   System.out.println( eP );
        }

        //2. ВЫПОЛНИТЬ ТЕСТЫ, НЕ ТРЕБУЮЩИЕ УЧАСТИЯ ЛЮДЕЙ
    Frame[] imageTests = {          //Показать совокупности тестовых данных
            new testCharacterImage(  //Нет рисунка
                new EncounterCharacter( «GuyWithNoImage», null ) ),
            new testCharacterImage(  //Есть рисунок
                new EncounterCharacter( «Elena», «elena.gif» ) )
        };

    for( int i = 0; i < imageTests.length; i++) {   //Показать окно каждого
                                                    //теста
            imageTests[i].setSize(400, 250);        //Подобрать размеры
            imageTests[i].setLocation((i * 110 + 50) & 500,
                (i + 60 + 25) % 200);
            imageTests[i].setVisible(true);
            imageTests[i].show();
        }

    try {                       //Пусть пользователь проанализирует окна
        Thread.currentThread().sleep(15*1000);
        } catch( Exception exc) {
        }

        for( int i = 0; i < imageTests.length; i++)   //Закрыть окна
            imageTests[i].dispose();

        System.exit(0);
}
```

```
/**  Тестирование комбинации методов класса
* @param        destinationP    Место записи результатов тестирования
* @exception    IOException     В случае невозможности открытия или
*                               доступа к destinationP
*/
public static void testEncounterCharacterClass( String destinationP )
    throws IOException
{   /* Подготовка к тесту */

    PrintWriter outM=new PrintWriter( new FileOutputStream(destinationP) );
    System.out.println(
     «\nEncounterCharacter class test results on » + destinationP + «\n» );

    /*
    * Будут тестироваться последовательности следующих методов:
    *
    *  a.  adjustQuality( String qualityP, float qualityValueP )
    *  d.  deleteFromEncounterCharacters(
                               EncounterCharacter encounterCharacterP )
    *  ge. EncounterCharacter getEncounterCharacter( String nameP )

    *  gq. float getQualityValue( String qualityP )

    *  gt. float getTolerance()
    *  io. int indexOf( String qualityP )
    *  ii. insertIntoEncounterCharacters(
                               EncounterCharacter encounterCharacterP )
    *  m.  int maxNumCharsInName()
    *  sq. setQuality( String qualityP, float qualityValueP )
    *  so. float sumOfQualities()
    * Часто встречаются следующие последовательности:
    *  ge-aq-so
    *  ge-sq-a-gq

    *  . . . . .
    * Ошибки наиболее вероятны в следующих последовательностях:
    *  ge-aq-aq-gq-so
    *  . . . . .
    */

    /* Тест C1: ge-aq-so */
    // Метод «ge»
    EncounterCharacter eC1M = new EncounterCharacter( «CharForTestC1" );
    eC1M.adjustQuality(QUAL_STRENGTH,  40.0f );                  // aq
    TestExecution.printReportToFile( outM,
        «Class test ge-aq-so», eC1M.sumOfQualities(), 100.0f );  // so
```

продолжение

Листинг 8.3 (*продолжение*)

```
    /* Тест C2: ge-aq-aq-gq-so */
    EncounterCharacter eC2M = new EncounterCharacter( «CharForTestC2" );//ge
    eC2M.adjustQuality(QUAL_STRENGTH,  40.0f );                    // aq
    eC2M.adjustQuality(QUAL_STAMINA,  20.9876f );                  // aq

    TestExecution.printReportToFile( outM,
            «Class test ge-aq-aq-gq-so: part 1",
            eC2M.getQualityValue( QUAL_STAMINA ),  20.9876f ); // gq

    TestExecution.printReportToFile( outM,
            «Class test ge-aq-aq-gq-so: part 2",
            eC2M.sumOfQualities(), 100.0f );                    // so

    /* ТЕСТИРОВАНИЕ ИНВАРИАНТОВ
    *   Проверить инвариант «qualValueI[i] >=0"
    *   после выполнения последовательностей методов, приведенных выше
    */
    boolean truthM = true;
    for( int i = 0; i < qualityTypeS.length; ++i )
    {   /* Присвоить truthM значение false,
        *  если хоть одно вхождение eC1M.qualValueI < 0
        */
        truthM = truthM && ( eC1M.qualValueI[i] >= 0.0f );

    }
    TestExecution.printReportToFile( outM,
        «Class test for the invariant 'qualValueI[i] >=0'», truthM, true );

    /* Завершение */
    outM.close();
    System.out.println( «\nТестирование класса EncounterChar завершено.» );

} // Конец testEncouterCharacterClass

/** Тестирование всех методов класса одновременно
* @param        destinationP    Место записи результатов тестирования
* @exception    IOException     В случае невозможности открытия или
*                               доступа к destinationP
*/
public static void testEncounterCharacterMethods( String destinationP )
    throws IOException
{   /* Подготовка к тестированию */
    FileWriter outM = new FileWriter( new File( destinationP ) );
    System.out.println( «EncounterCharacter method test results on »
        + destinationP + «\n» );
```

```
    /* Тестирование getEncounterCharacter() */

    // На входе корректное имя
    EncounterCharacter eCNorM = new EncounterCharacter( «qwerty» );
    TestExecution.reportToFile( outM,
        «GetCharacter Test 1: nominal value», eCNorM.getName(), «qwerty» );

    // На входе null
    EncounterCharacter eCNullM = new EncounterCharacter( null );
    TestExecution.reportToFile( outM, «GetCharacter Test 2: null parameter»,
        eCNullM.getName(), GameCharacter.DEFAULT_NAME );

    // Слишком длинное имя
    String tooLongM =
        «12345678901234567890123456789012345678901234567890";
    EncounterCharacter eCTooLongM =  new EncounterCharacter(tooLongM);
    TestExecution.reportToFile( outM,
        «GetCharacter Test 3: Limit parameter values. »
            + «max name len = » + eCTooLongM .maxNumCharsInName(),
        eCTooLongM.getName(),
        tooLongM.substring(0, eCTooLongM.maxNumCharsInName()) );

    // Имя нулевой длины
    EncounterCharacter eCZeroM = new EncounterCharacter( «» );
    TestExecution.reportToFile( outM,
        «GetCharacter Test 4: zero-length»,
        eCZeroM .getName(), GameCharacter.DEFAULT_NAME );

// Недопустимые символы
    EncounterCharacter eCPuncM = new EncounterCharacter( «a+b» );
    TestExecution.reportToFile( outM,
        «GetCharacter Test 5: bad char '+' »,
        eCPuncM .getName(), GameCharacter.DEFAULT_NAME );

    /* Проверка indexOf() на правильность имен всех характеристик */
    for( int i = 0; i < qualityTypeS.length; ++i )
        try { TestExecution.reportToFile( outM,
            «indexOf() Test 1.» + i + «: valid name: » +
            qualityTypeS[i], indexOf(qualityTypeS[i]), i );
            } catch( Exception eP )
            { TestExecution.reportToFile( outM,
            «indexOf() Test 1: valid name: compare »,
            «indexOf('» + qualityTypeS[i] + «')»,
            «with expected » + i );
            }
```

продолжение

Листинг 8.3 (*продолжение*)

```
/* Проверка indexOf() на неправильное имя характеристики */
try { TestExecution.reportToFile( outM,
        «indexOf() Test 2: invalid name: zorch», indexOf(«zorch»), -1 );
    } catch( Exception eP )
    { TestExecution.reportToFile( outM,
    «indexOf() Test 2: valid name: compare »,
    «indexOf(\»zorch\")", «with expected -1" );
}

/* Тестирование setQuality() */

// Подготовка к тестированию
EncounterCharacter hank = new EncounterCharacter( «Hank» );


// Номинальное значение
hank.setQuality(QUAL_STRENGTH , 10.3f );
TestExecution.reportToFile( outM,
    «setQuality() Test 1: nominal value»,
    hank.getQualityValue( QUAL_STRENGTH ), 10.3f );

// Значение вне диапазона
hank.setQuality( QUAL_PATIENCE, -6.2f );
TestExecution.reportToFile( outM,
    «setQuality() Test 2: nominal value»,
    hank.getQualityValue(QUAL_PATIENCE ), 0.0f );

// Значение ниже минимального разрешенного
hank.setQuality( QUAL_STAMINA, getTolerance() * 0.9f );
TestExecution.reportToFile( outM,
    «setQuality() Test 3: value close to zero»,
    hank.getQualityValue(QUAL_STAMINA), 0.0f );

// Тестирование adjustQuality().

// Подготовка к тестированию и проверка:
// все значения характеристик должны быть равны 20.
EncounterCharacter harvey = new EncounterCharacter( «Harvey» );
TestExecution.reportToFile( outM,
    «adjustQuality() test 0: verify that values add to 100",
    harvey.sumOfQualities(),  100.0f );

// Номинальная установка:
// сила = 30, остальные характеристики по 70/4 каждая
harvey.adjustQuality(QUAL_STRENGTH , 30.0f );
TestExecution.reportToFile( outM,
```

```
        «adjustQuality() test 1: values sum to 100 after adjusting»,
        harvey.sumOfQualities(), 100.0f );
    TestExecution.reportToFile ( outM,
        «adjustQuality() test 2: values adjusted as commanded»,
        harvey.getQualityValue(QUAL_STRENGTH ), 30.0f );

    // Установка, приводящая к нулевому значению
    harvey.adjustQuality( QUAL_STAMINA, 99.0f );
    TestExecution.reportToFile( outM,
        «adjustQuality() test 3: verify low value reverts to zero»,
        harvey.getQualityValue( QUAL_STRENGTH ), 0.0f );

    // Завершение
    outM.close();
    System.out.println( «\nMethod tests of EncounterCharacter concluded.» );
}

/** Класс для тестирования перерисовки персонажей.
* Создает окно, содержащее несколько копий изображения персонажа.
*/

private static class testCharacterImage extends Frame
{
    /** Атрибут объекта, запоминающий, какое изображение персонажа выводить*/
    private EncounterCharacter characterI;

    /** Основной конструктор: создает окно для тестирования некоторого
    *изображения персонажа.
    * @param    characterP   Персонаж, чье изображение должно тестироваться
    */

    testCharacterImage(EncounterCharacter characterP)
    {
        super(characterP.getName()); //Обычная инициализация
        characterI = characterP;     //Запомнить, какой персонаж тестируем
    }

/** Перерисовывает область изображения во фрейме.
*@param   drawP   Графический контекст для рисования персонажа
*/
public void paint(Graphics drawP)
{
    Dimension frameSizeM = getSize(); //Размер области окна
    int widthUnitM = frameSizeM.width / 5; //Удобное разбиение окна
    int heightUnitM = frameSizeM.height / 5;
```

продолжение

Листинг 8.3 (*продолжение*)

```
    //Нарисовать маленьким, лицом направо
    characterI.showCharacter(this, drawP,
        new Point(widthUnitM, heightUnitM), heightUnitM, false);

    //Нарисовать большим, лицом налево
    characterI.showCharacter(this, drawP,
        new Point(widthUnitM*4, heightUnitM*3), heightUnitM*2, true);

    //Нарисовать большим, лицом направо
    characterI.showCharacter(this, drawP,
        new Point(widthUnitM*2, heightUnitM*2), heightUnitM*2, false);

    //Нарисовать маленьким, лицом налево
    characterI.showCharacter(this, drawP,
        new Point(widthUnitM*3, heightUnitM*4), heightUnitM, true);
    }
} // Конец testCharacterImage
```

Глава 9
Интеграция, верификация и валидация системы

Содержание этой главы в контексте процесса разработки программного обеспечения показано на рис. 9.1.

- ✦ Разделы 9.1–9.8
- ✦ Упражнения
- ✦ Примеры
  1) Пример 1. План управления конфигурациями программного обеспечения (SCMP). Приложение A: План создания базиса интеграции
  2) Документация по тестированию программного обеспечения (STD) для видеоигры *Встреча*

Рис. 9.1. Схема разработки программного обеспечения: темы главы 9

Вопросы, рассматриваемые в этой главе:

- Планирование интеграции модулей системы.
- Понимание необходимых типов тестирования.
- Планирование и выполнение тестирования.

  На уровне всей системы в целом.

9.1. Введение

9.1.1. Значение фазы интеграции

Поскольку программные продукты довольно сложны по своей структуре, их формируют из частей, которые создаются независимо, а затем собираются в единое целое. Интеграция как раз и относится к процессу сборки. Различные виды тестов проводятся как над частично собранным приложением, так и над всем продуктом в целом.

Фаза интеграции водопадного процесса часто преподносит неприятные сюрпризы, связанные с несовместимостью интегрируемых частей. По этой причине USDP, в частности, старается ибежать сборки большого количества элементов, благодаря последовательной интеграции с помощью многочисленных итераций. Закрашенные области на рис. 9.2 показывают, что интеграция на самом деле имеет место и на итерациях конструирования и перехода [64].

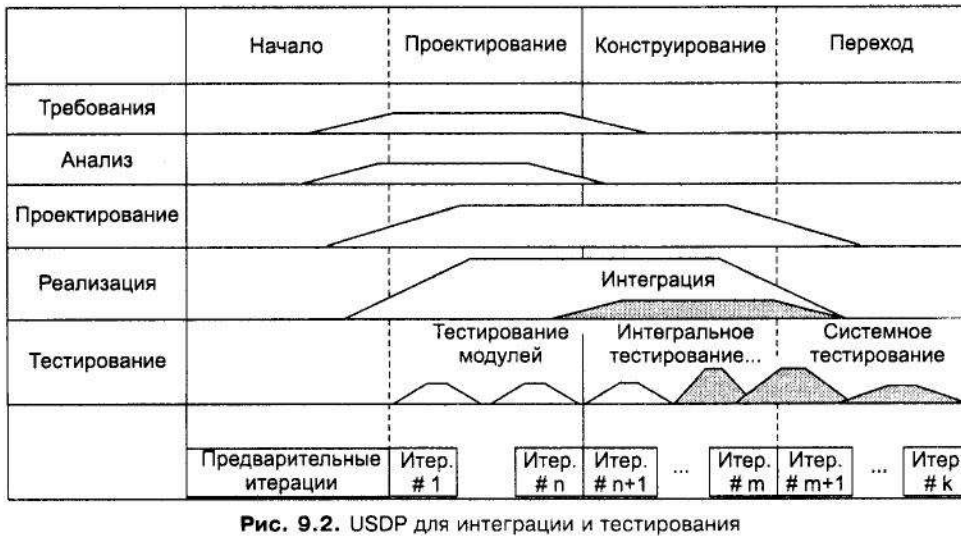


Рис. 9.2. USDP для интеграции и тестирования

При переходе от одной стадии процесса разработки к другой достаточно вероятна потеря информации. Отмеченные Майерсом [82] места в водопадном

процессе, в которых обычно происходят потеря информации и утрата понимания, показаны на рис. 9.3.

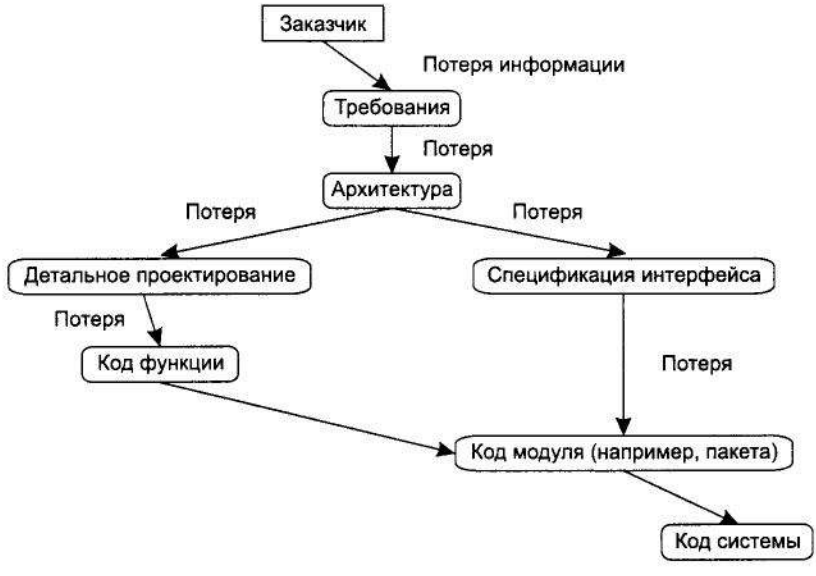


Рис. 9.3. Схема разработки

Во избежание этих потенциально возможных потерь информации используется непрекращающееся тестирование и интеграция. Несмотря на то что тестирование модулей как части целого приложения в конечном окружении имеет огромное значение (этому виду тестирования посвящена данная глава), оно не заменяет тщательного тестирования каждого модуля в отдельности перед добавлением его к системе (см. главу 8).

9.1.2. Верификация, валидация и системное тестирование

Вспомните, что *верификация* позволяет определить, правильно ли мы создаем приложение. Другими словами, действительно ли мы на текущей фазе создаем именно те артефакты, что были специфицированы на предыдущей фазе? Применительно к интеграции верификация приравнивается к подтверждению того, что мы собираем именно те компоненты, которые мы планировали собрать, и именно тем способом, каким это было запланировано. Такая проверка может быть произведена при помощи инспектирования результатов интеграции.

Валидация позволяет выяснить, правильный ли результат у нас получается. Другими словами, удовлетворяет ли наш продукт требованиям, изложенным в SRS? На фазе интеграции этого добиваются с помощью системного тестирования.

После завершения сборки, итерации или всего приложения тщательное тестирование требует, чтобы мы сначала выполнили модульные тесты функций (методов) и модулей (классов или пакетов). В этот раз, однако, эти тесты следует пройти в некотором контексте, а не изолированно друг от друга. Здесь требуется меньше драйверов и заглушек, что приводит к меньшему количеству сложностей и ошибок. Если мы тестируем финальную сборку, то нам вообще не следует использовать драйверы или заглушки. Вся разница между автономными модульными тестами и модульными тестами, выполняемыми в контексте системы, показана на рис. 9.4, где модулями считаются функции.

Рис. 9.4. Тестирование модулей в контексте

Например, модульное тестирование метода `adjustQuality()` в классе *СредаВстречи* выполнялось с использованием тестового метода `testEncounterCharacterMethods()`, добавленного в класс *СредаВстречи* (см. пример в главе 8). С другой стороны, тестирование `adjustQuality()` в контексте готовой программы выполняется путем запуска программы таким образом, чтобы убедиться, что метод неоднократно вызывался. Это можно сделать путем периодического открытия окна во время игры и изменения значений характеристик.

Движение артефактов (преимущественно документов и кода) между стадиями проекта и между тестами разных типов [82] показано на рис. 9.5. Модульные тесты и тесты функций выполняются в двух разных режимах. В первый раз они выполняются изолированно, как модульные тесты. Второй раз они выполняются в контексте всей программы. По этой причине они пронумерованы дважды.

Связь запускаемых тестов с документацией показана на рис. 9.6. Вспомните, что валидация — это процесс, в результате которого мы хотим убедиться, что мы создаем «правильную» программу, и поэтому такие тесты проводятся согласно исходным требованиям. Другие тесты проверяют, что программа создается так, как мы намеревались, что является процессом верификации. Например, тесты интерфейса проверяют, точно ли реализация отражает запланированные интерфейсы.

Упомянутые на рис. 9.6 тесты будут кратко описаны далее, а затем тщательно разработаны в последующих разделах главы.

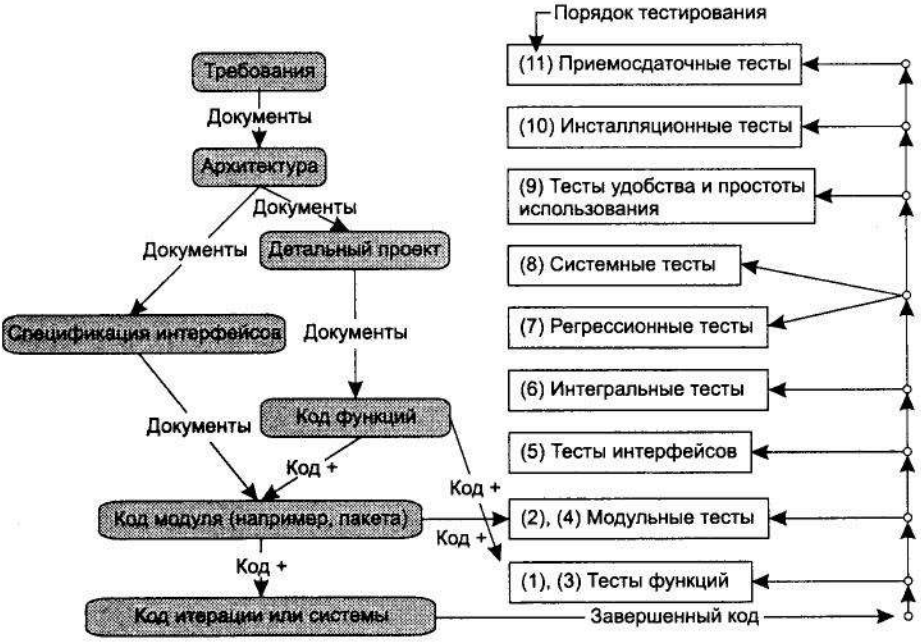


Рис. 9.5. Обзор тестирования: поток артефактов

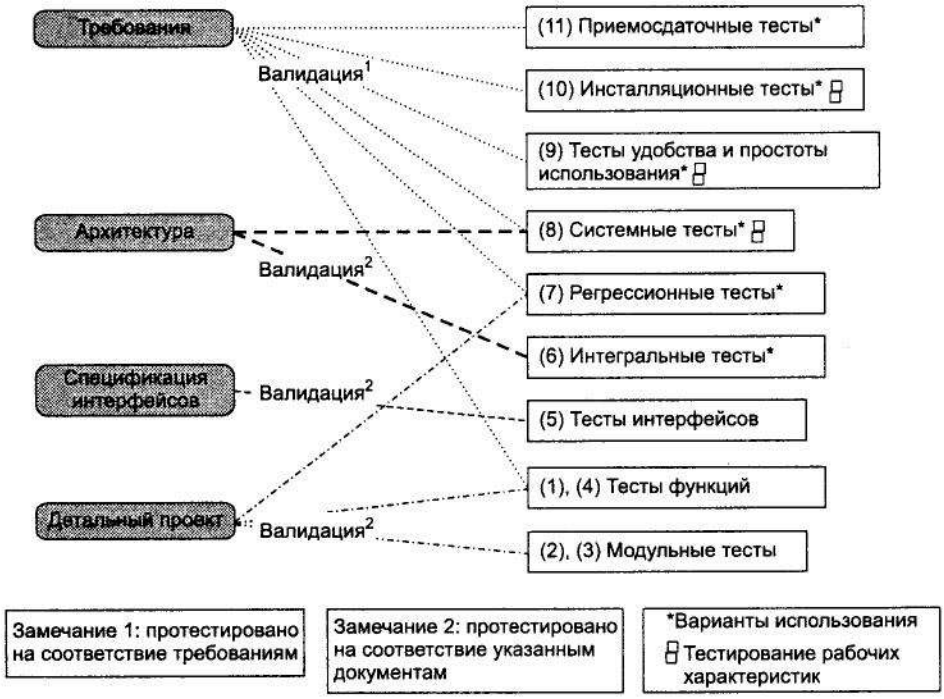


Рис. 9.6. Тестирование для верификации и валидации

Когда код системы интегрирован или частично интегрирован (нижняя часть рис. 9.5), становится возможным протестировать части в контексте всей системы вместо использования автономного подхода. Чтобы сфокусировать тестирование на разработанных частях программы, нам придется продумать подходящие входные данные.

1. Нам пришлось создать заглушки и драйверы для выполнения модульного тестирования *функций* и *классов*, учитывая возможность существования ошибок и неполного охвата. Если их нельзя оставить в коде по организационным причинам или в связи с ограниченностью ресурсов, этот код можно отложить в сторону с возможным использованием в будущем. Альтернатива заключается в добавлении или исключении тестового кода посредством условной компиляции (присоединить или исключить код модульного тестирования).
2. Аналогичным образом становится возможным повторно протестировать другие *модули* (например, пакеты) в контексте системы.
3. Тестирование *интерфейсов* подразумевает повторную валидацию интерфейсов между модулями.
4. Цель *регрессионного тестирования* заключается в проверке того, что добавления к системе не уменьшили ее возможностей. Другими словами, регрессионное тестирование проводится согласно требованиям, которые уже были выполнены перед добавлением новых возможностей. Только когда артефакт прошел регрессионное тестирование, мы будем готовы тестировать работу добавленного кода.
5. *Интегральное тестирование* выполняется над частично сконструированной системой для проверки того, что результат интеграции дополнительных программ (например, классов) работает, как запланировано. Например, мы можем реализовать сначала пакет *СредаВстречи* и тщательно его протестировать. Затем мы можем реализовать пакет *РолиВстречи*. После интеграции этого пакета в пакет *СредаВстречи* мы выполняем интегральное тестирование, проверяя, что персонажи должным образом ведут себя в зонах.
6. *Системное тестирование* выполняется над программой в целом или над разработанными версиями.

   Системные и интегральные тесты проводятся в соответствии с архитектурой. Другими словами, они проверяют, чтобы программа следовала разработанной архитектуре и чтобы архитектура работала должным образом. Например, архитектура видеоигры *Встреча* разработана таким образом, чтобы при вступлении внешнего персонажа в зону, в которой находится персонаж игрока, генерировалось событие в пакете *СхемаВстречи*. Событие обрабатывается пакетом *ИграВстреча*. В некоторых случаях внешний персонаж может исчезать, что должно найти отражение в пакете *РолиВстречи*. *Системные* тесты проверяют такое поведение.

   Системное тестирование также валидирует требования, как функциональные, так и нефункциональные. Нефункциональные требования включают в себя требования к рабочим характеристикам, таким как скорость работы и использование ресурсов.

7. *Тестирование удобства и простоты использования* валидирует приемлемость программы для ее конечных пользователей.
8. *Тестирование инсталляции* выполняется при установке программы на целевых платформах.
9. *Приемосдаточные тесты* выполняются клиентом для валидации приемлемости программы.

Далее мы подведем итоги и обсудим типы тестирования более подробно.

9.2. Процесс интеграции

9.2.1. Описание интеграции

Возможный процесс интеграции для первой итерации постройки подвесного моста (одноуровневая версия), а также для второй итерации (двухуровневая версия) приведены на рис. 9.7. Каждая итерация связана с этапом постройки. Спланирована точная последовательность действий по созданию *сборок*, которая завершает итерацию.

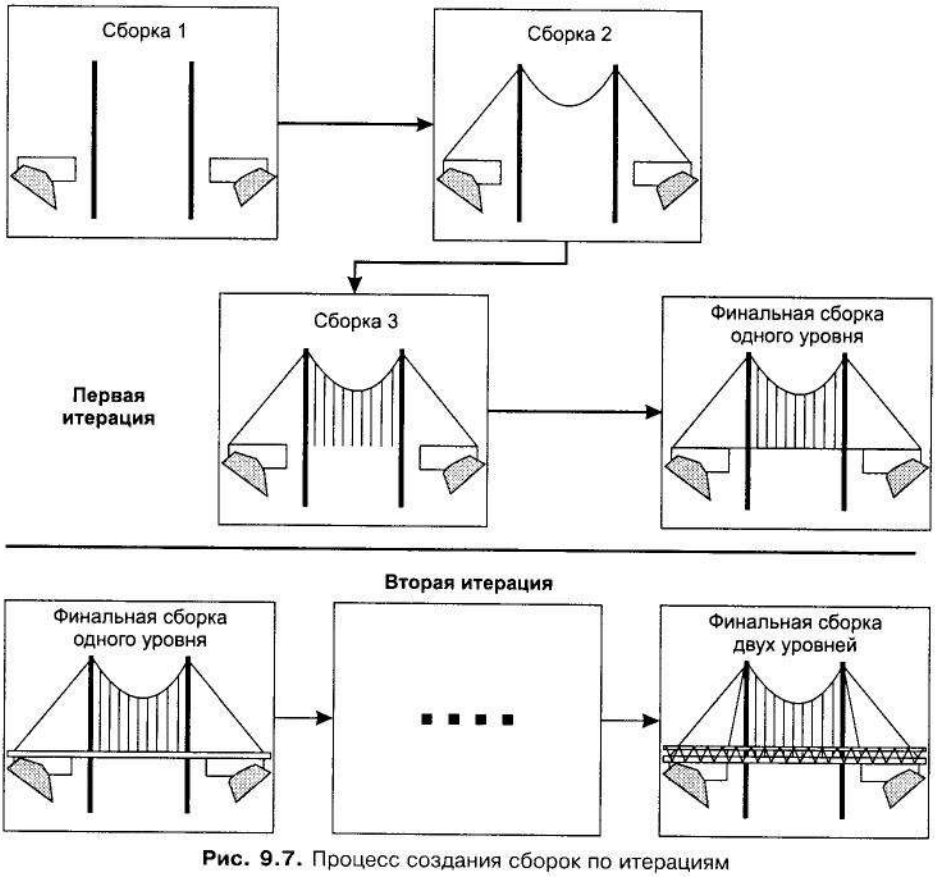


Рис. 9.7. Процесс создания сборок по итерациям

Простейший вид интеграции состоит из добавления новых элементов к базису (существующему коду) на каждой итерации по спирали (рис. 9.8). Фаза реализации состоит из кодирования новых частей, после которого эти новые части интегрируются в базис.

Рис. 9.8. Интеграция в спиральной разработке

Процесс интеграции кода требует не меньшего искусства и навыков, чем процесс интеграции физических объектов. Он может быть очень сложным. Как и в нашем примере с мостом, каждая программная итерация разбивается на стадии. Для USDP [64] это разбиение представлено на рис. 9.9.

На рис. 9.9 показаны группы итераций (например, итерации конструирования), где каждая итерация разбита на несколько сборок. Такая организация существенно важна для крупных проектов.

По завершении разработки архитектуры важно определить легкость, с которой части будут интегрироваться в проект. В отличие от некоторых физических разработок, в нашем случае редко удается завершить отдельные программные модули до их интеграции в проект. Одна причина этого заключается в том, что типовые программные модули выполняют обслуживание нескольких клиентов, в то время как физические модули обслуживают очень ограниченное число «клиентов», а зачастую вообще всего один. Например, каждая опора моста поддерживает лишь одну или две секции дороги. Кроме того, когда программные требования более понятны, становятся очевидны и новые клиенты для каждого модуля. Таким образом, программные сборки часто бывает необходимо интегрировать в частично сконструированные модули, как в «типовой» последовательности, а не как в «модульно-ориентированной» (рис. 9.10).

Рис. 9.9. Связь между сборками и итерациями в USDP

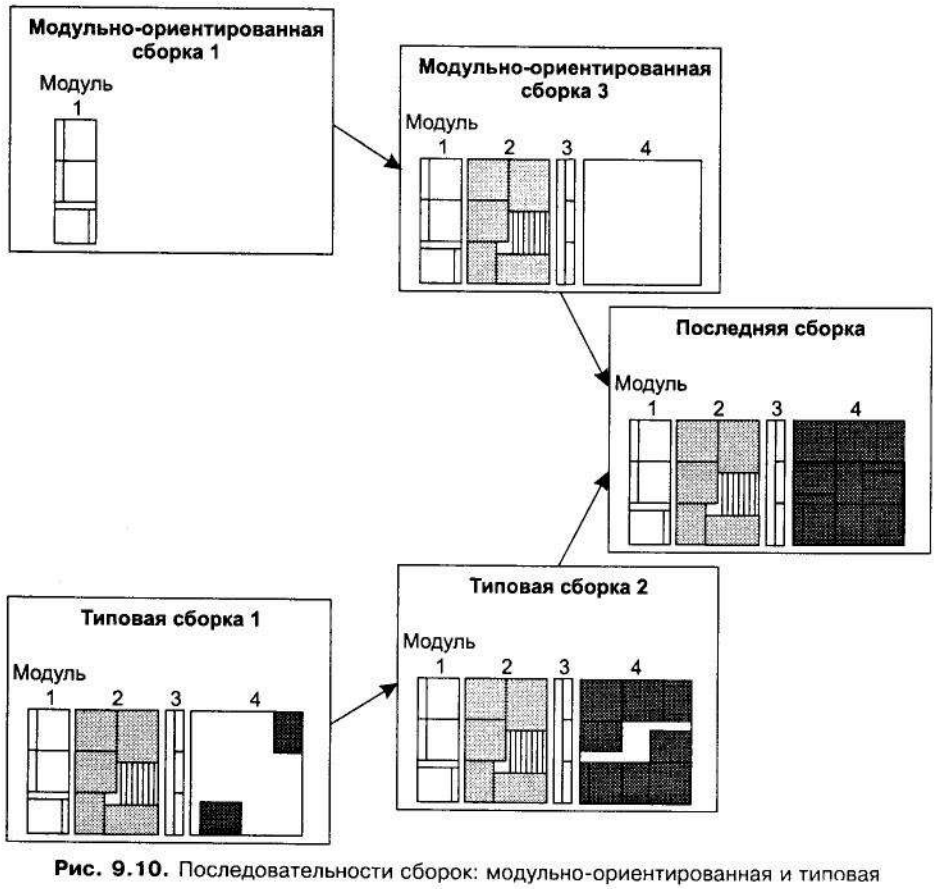


Рис. 9.10. Последовательности сборок: модульно-ориентированная и типовая

Хотя типовой процесс сборки имеет недостаток, заключающийся в работе с незавершенными модулями, он имеет и преимущество, состоящее в выполнении интеграции на ранних стадиях процесса разработки. Это помогает уменьшить риск, связанный с интеграцией завершенных крупных модулей.

Важность разработки проекта модулей возрастает благодаря сложности интеграции программ (таких, как классы и пакеты). Они должны быть как можно лучше сосредоточены на своей цели, а их взаимные интерфейсы должны быть как можно более узкими. Эти вопросы, касающиеся сцепления и связности, подробно обсуждались в разделе 5.1.4.

ОДИН ИЗ СПОСОБОВ ПЛАНИРОВАНИЯ ИНТЕГРАЦИИ И СБОРОК

1. Поймите декомпозицию архитектуры.
   - Постарайтесь сделать архитектуру простой для интеграции.
2. Определите части архитектуры, которые будут реализовываться на каждой итерации:
   - сначала соберите каркасные классы или делайте это параллельно;
   - по возможности интегрируйте «последовательно»;
   - соберите достаточное количество пользовательских интерфейсов для привязки тестирования;
   - документируйте требования для каждой итерации;
   - постарайтесь выполнять сборки снизу вверх хотя бы в течение некоторого промежутка времени, чтобы части системы были доступны, когда понадобятся;
   - старайтесь планировать итерации, чтобы уменьшить риск, в первую очередь уделив внимание самому крупному риску;
   - определите итерации и сборки, чтобы каждый вариант использования обрабатывался отдельно.
3. Разложите каждую итерацию на сборки, если нужно.
4. Спланируйте процессы тестирования, пересмотра и инспектирования.
5. Обновляйте расписание проекта, отображая в нем результаты.

Указанные этапы будут обсуждаться более подробно далее в тексте этой главы.

Тестирование упрощается после объединения всех реализаций вариантов использования в каждой сборке вместо тестирования частей вариантов использования. Разрабатывая относительно небольшие варианты использования, вы, прежде всего, упрощаете процесс добавления их в сборку. Поскольку пользовательские интерфейсы рано или поздно все равно придется тестировать, желательно, чтобы их самих или их ключевые части можно было встроить как можно раньше, тем самым обеспечив возможность тестирования развивающейся программы. Альтернативой является сборка временных интерфейсов для использования во время интегрального тестирования.

Якобсон [63] отметил, что в общем случае в качестве руководства к планированию сборок проще использовать развитие сборок снизу вверх. Согласно этому подходу, части создаются перед их использованием для конструирования более крупных модулей. Восходящий процесс можно успешно скомбинировать с реализацией классов структур, которая является нисходящим процессом.

9.2.2. Типичная схема процессов интегрального и системного тестирования

Типичная последовательность действий для интеграции программной системы включает следующие шаги:

1. Определить границы всех тестов.
2. Для каждой итерации...
   1) Для каждой сборки...
      - Выполнить регрессионное тестирование для предыдущей сборки.
      - В случае необходимости повторно протестировать функции.
      - В случае необходимости повторно протестировать модули.
      - В случае необходимости повторно протестировать интерфейсы.
      - Выполнить интегральное тестирование сборки (раздел 9.3.7).

      Разработка итерации завершена.
   2) Выполнить для итерации системные тесты и тесты удобства и простоты использования (разделы 9.3.4, 9.3.5).

   Система реализована.
3. Выполнить тестирование инсталляции (раздел 9.3.8).

   Система установлена.
4. Выполнить приемосдаточное тестирование (раздел 9.3.7).

   Работа завершена.

Пакеты, выбранные для видеоигры *Встреча*, а также выбранные классы предметной области показаны на рис. 9.11. Интеграция этих пакетов состоит из их сборки по этапам, в результате которой должна получиться завершенная программа. Это выполняется в соответствии с планом интеграции, который можно документировать в SCMP. Это пример необходимости постоянного обновления документов. Вспомните, что SCMP является первым документом проекта. Мы возвращаемся к нему (в SDD) после определения архитектуры для определения последовательности интеграции.

Существенными факторами, определяющими последовательность интеграции, являются следующие:

- Технические факторы:
  - использование модулей другими модулями. Собирайте и интегрируйте используемые модули до модулей, использующих их;
  - определение и использование классов каркаса.
- Уменьшение риска:
  - раннее выполнение интеграции;
  - как можно более раннее выполнение частей программы, порождающих основные риски.
- Требования:
  - показ частей или прототипов заказчикам.

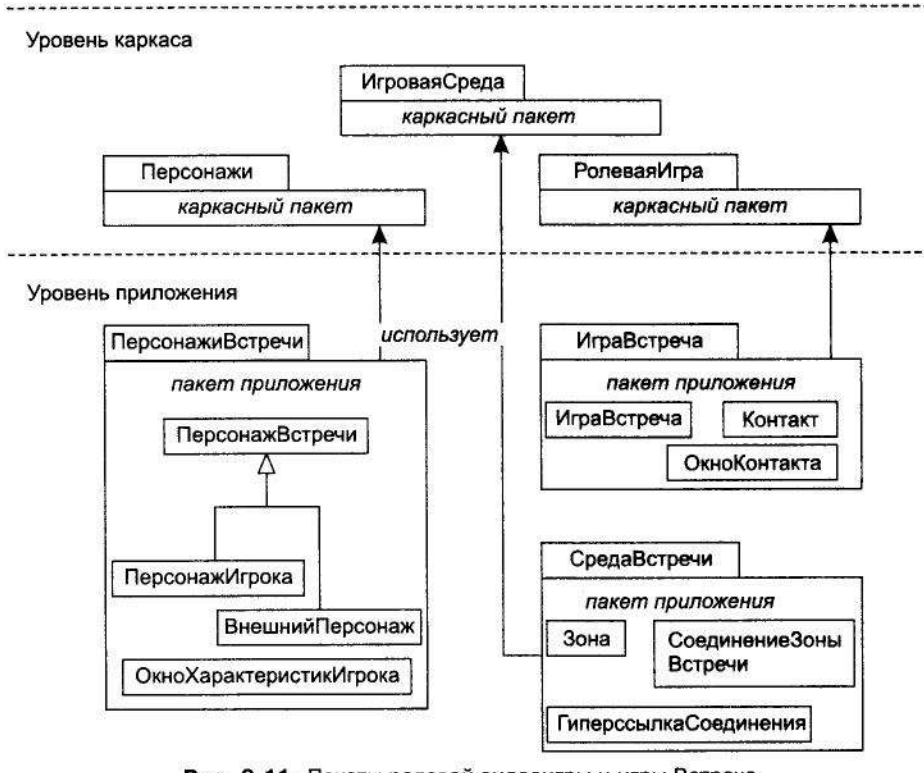


Рис. 9.11. Пакеты ролевой видеоигры и игры Встреча с отмеченными классами предметной области

Их порядок зависит от требований проекта. В рискованных проектах мы будем стремиться интегрировать рискованные части как можно скорее, чтобы оценить эффективность нашего проектирования. Показ отдельных частей программы заказчику также диктует нам порядок интеграции. В противном случае мы будем интегрировать использованные модули до модулей, использующих их, тем самым сводя к минимуму использование временного кода драйвера.

Поскольку пакет *ИграВстреча* в видеоигре *Встреча* использует (ссылается на) пакеты *СредаВстречи* и *ПерсонажиВстречи*, мы в первую очередь интегрируем последние два пакета. После этого игровой драйвер будет интегрирован, что позволит выполнить игру. Результирующий план интеграции показан на рис. 9.12.

Поскольку у нас есть только три программных пакета, которые мы должны интегрировать в игру, и поскольку наша игра является лишь прототипом, план интеграции довольно прост. Он состоит из двух итераций, разбитых на три сборки. Поскольку то, что мы создадим, будет лишь началом настоящей видеоигры, мы, возможно, захотим описать план интеграции в терминах USDP. Итак, план

на рис. 9.12 показывает начальную итерацию, состоящую из двух сборок. Дальнейшие итерации пока еще не показаны.

Рис. 9.12. План интеграции для видеоигры Встреча

Альтернативный план интеграции для видеоигры *Встреча* показан на рис. 9.13. Этот план осуществляется поэтапно по классам из различных пакетов. Он не имеет целью закончить разработку пакета и затем интегрировать его с другими изолированно разработанными пакетами. Вместо этого в каждой сборке добавляются классы к нескольким пакетам. Достоинство этого плана интеграции заключается в том, что он интегрирует классы по ходу работы, стараясь избежать катастрофических ошибок, которые часто случаются при интеграции «готовых» пакетов.

9.3. Процесс тестирования

9.3.1. Интегральное тестирование

Интегральное тестирование проверяет каждую сборку интеграции и каждую итерацию. Способ, которым можно спланировать и выполнить тестирование интеграции наряду с регрессионным и модульным тестированием, показан далее. Эти концепции объясняются позднее в этой главе.

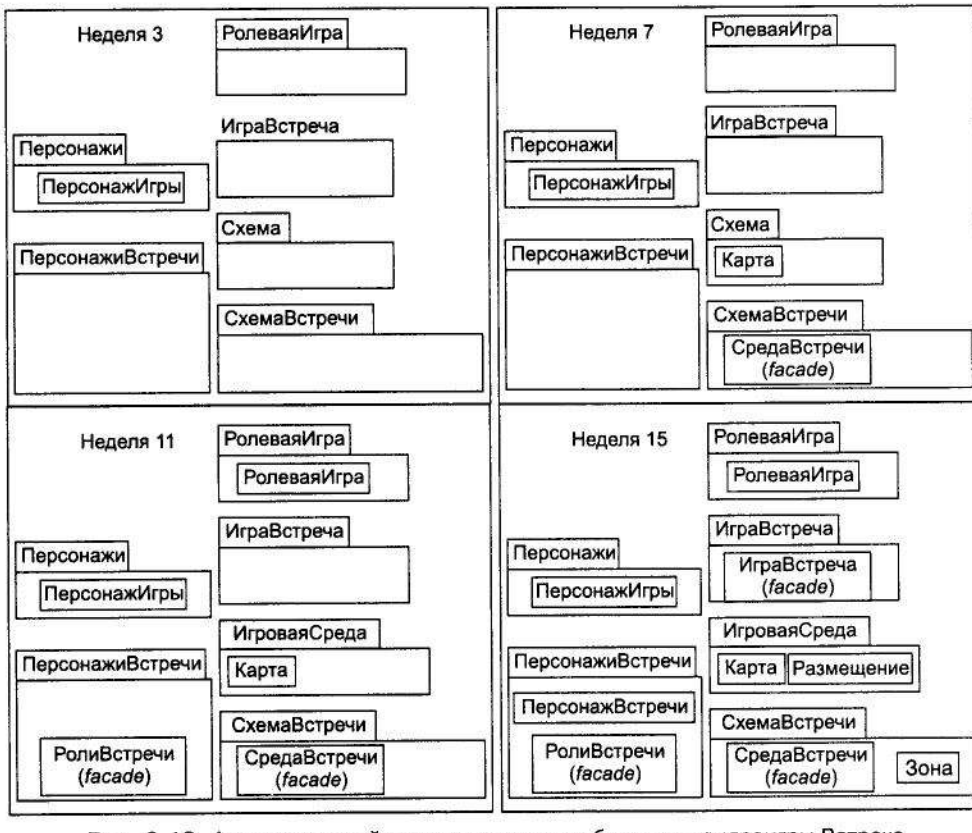


Рис. 9.13. Альтернативный процесс создания сборки для видеоигры Встреча

ОДИН ИЗ СПОСОБОВ ПЛАНИРОВАНИЯ И ВЫПОЛНЕНИЯ ИНТЕГРАЛЬНЫХ ТЕСТОВ

1. Решите, как и где хранить, повторно использовать и кодировать интегральные тесты.
   - Отразите это в расписании проекта.
2. Выполните столько модульных тестов (снова), сколько позволит время:
   - на этот раз в контексте сборки.

     При этом не требуются драйверы или заглушки;
   - расставьте приоритеты тем модулям, в которых вероятнее всего могут появиться ошибки.
3. Выполните регрессионные тесты.

   Убедитесь, что существующие возможности системы не были нарушены.
4. Убедитесь, что требования сборки определены должным образом.
5. Выполните варианты использования, которые должны быть реализованы в сборке.
   - Протестируйте на соответствие SRS.
6. Выполните системные тесты для этой сборки.

Варианты использования являются идеальным источником тестовых вариантов для интегральных тестов. Как упоминалось выше, Якобсон [63] рекомендует

согласовывать каждый вариант использования с одной сборкой. Идея в том, чтобы варианты использования строились на основе уже интегрированных частей, тем самым формируя представительные тесты использования программы (рис. 9.14).

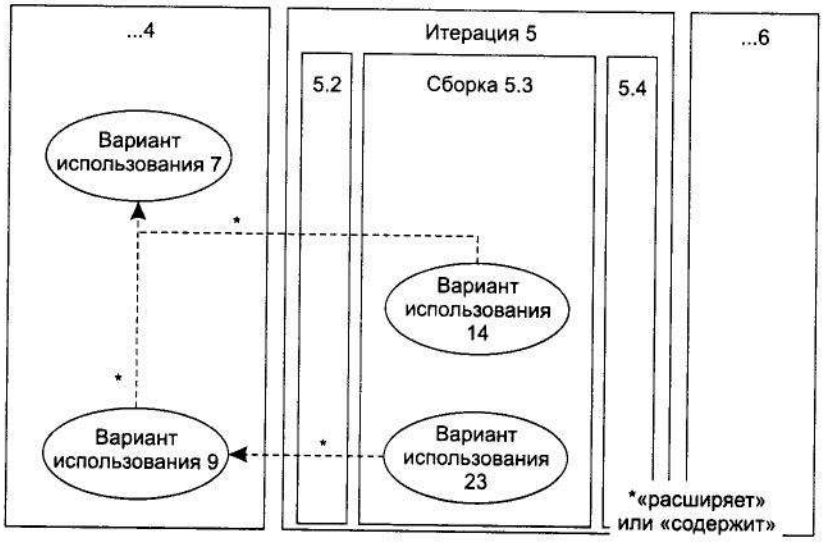


Рис. 9.14. Связь между вариантами использования, итерациями и сборками

Для адекватной оценки программы требуется большое количество тестов, и необходимо использовать их методичную организацию. Один из стилей организации тестовых вариантов заключается в упаковке их в классы, специально созданные для тестирования. Например, тест пакета можно разработать как класс, принадлежащий пакету. Класс или, может быть, весь тестовый пакет можно посвятить тестированию целой программы. Это можно показать как значки артефактов тестирования на модели реализации, обсуждавшейся в разделе 7.1.4.

Обычно сборки состоят из кода нескольких разработчиков, поэтому возникает много проблем при интеграции кода для создания сборки. По этой причине мы стараемся начать интеграцию и интегральное тестирование на ранних этапах процесса разработки, что позволит выполнять код в его первичном контексте.

Интегральное тестирование выполняется во время разработки сборки. Такие тесты обычно состоят из регрессионных тестов с дополнительным тестированием для утверждения новых добавлений. Считается непрактичным постоянно запускать полные формальные интегральные тесты. Следовательно, часто на постоянной основе применяются масштабированные неформальные интегральные тесты: они утверждают только, что система вроде бы работает так, как должна. Иногда такие тесты называют *«проверкой на дым»*. «Проверка на дым» либо убеждает программистов в том, что можно продолжать работать в том же духе (если не возникло никаких проблем), либо указывает на проблемы, которые могут привести к значительным задержкам на этапе интеграции.

План интеграции часто принимает форму, показанную на рис. 9.15 для примера банковской программы. По мере того как задача построения модулей

подходит к концу, модули по очереди интегрируются в основу (например, сливаются с основным продуктом). В этом случае процесс интеграции происходит между неделями 23 и 41.

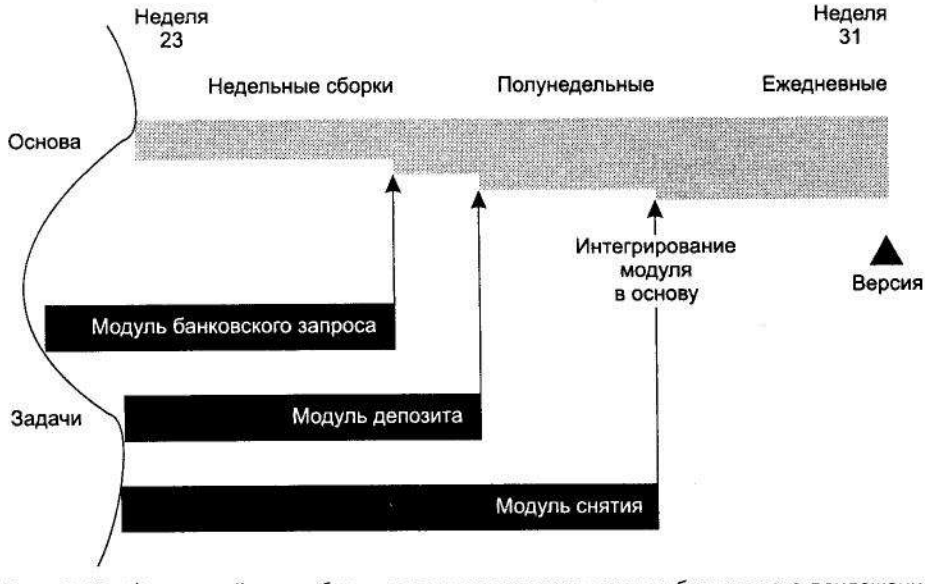


Рис. 9.15. Финальный план сборки кода и интеграции: пример банковского приложения

Процесс компиляции и тестирования частичных сборок нередко выполняется за ночь, и на время компиляции и тестирования разработка замораживается (рис. 9.16).

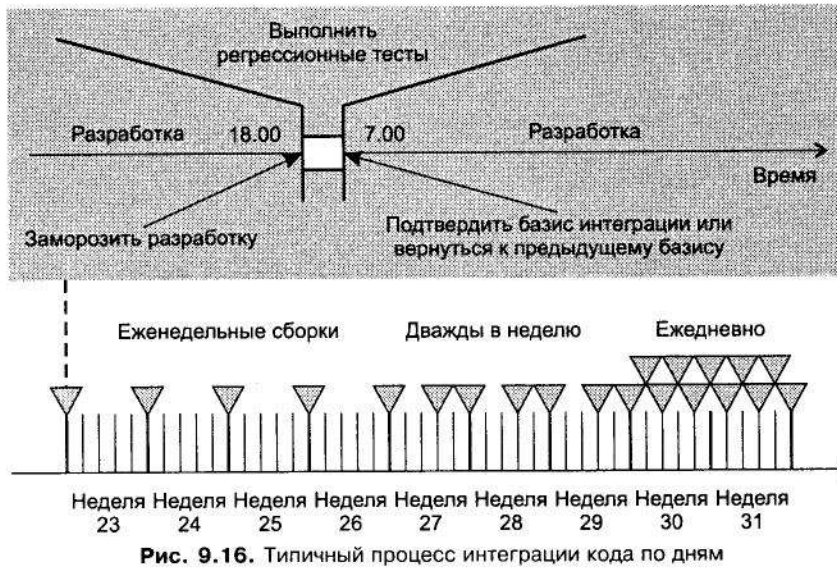


Рис. 9.16. Типичный процесс интеграции кода по дням

По мере достижения сроков выхода версии частота регрессионных тестов возрастает до тех пор, пока они не будут выполняться ежедневно, обычно ночью (см. рис. 9.16). Если регрессионное тестирование показывает, что существовавшая функциональность все еще имеет место, интегрированный код становится частью основы системы. Но если регрессионные тесты показали, что добавленный код порождает ошибки в существовавшей ранее функциональности, можно принять решение вернуться к базе ранее написанного кода, существовавшего до интеграции нового материала (эффективная «дезинтеграция»). Такой вид планирования ежедневных интегральных и регрессионных тестов был предложен в [21] и использовался, например, фирмой Microsoft.

9.3.2. Сотрудники, участвующие в тестировании, и артефакты

В этом разделе мы рассмотрим артефакты, связанные с процессом интегрального тестирования, согласно USDP. Они включают в себя следующее.

- *Модель вариантов использования* — набор вариантов использования, описывающих типичное использование программы и диаграммы последовательности, подробно описывающие их.
- *Тестовые варианты* — входные данные для каждого теста.
- *Процедуры тестирования* — способ, которым следует создавать и проводить тесты и оценивать результаты. Это могут быть процедуры с ручным управлением либо использующие инструменты автоматизации тестирования.
- *Оценка тестов* — подведение итогов, подробности и выгода от найденных ошибок.
- *План тестирования* — общий план руководства тестированием, в том числе порядок тестирования.
- *Компоненты теста* — исходный код самих тестов и программный код, который следует протестировать.
- *Дефекты*: отчет об обнаруженных в результате процесса дефектах, классифицированных по серьезности и типу.

Процесс интегрального тестирования в USDP требует работы таких специалистов, как тестовый инженер, разработчик компонентов, тестер интеграции и системный тестер. Их обязанности [64] показаны на рис. 9.17. Большинство артефактов, показанных на рис. 9.17, учитываются в IEEE-стандарте документации по тестированию, обсуждаемом далее.

9.3.3. Тестирование интерфейсов

Многие ошибки в программах связаны с проблемами интерфейсов между компонентами. При беспорядочном исполнении проекта группы находят более простым общение друг с другом, а не группы с группой. Вследствие этого проект и программные интерфейсы с легкостью могут быть поняты неправильно. Раздел определения интерфейсов в SDD является «Библией» для внутренних интерфейсов

программного приложения. Когда модули разработаны, можно приступать к тестированию интерфейсов. Оно осуществляется посредством генерации трафика через каждый интерфейс, обычно в форме функциональных вызовов.

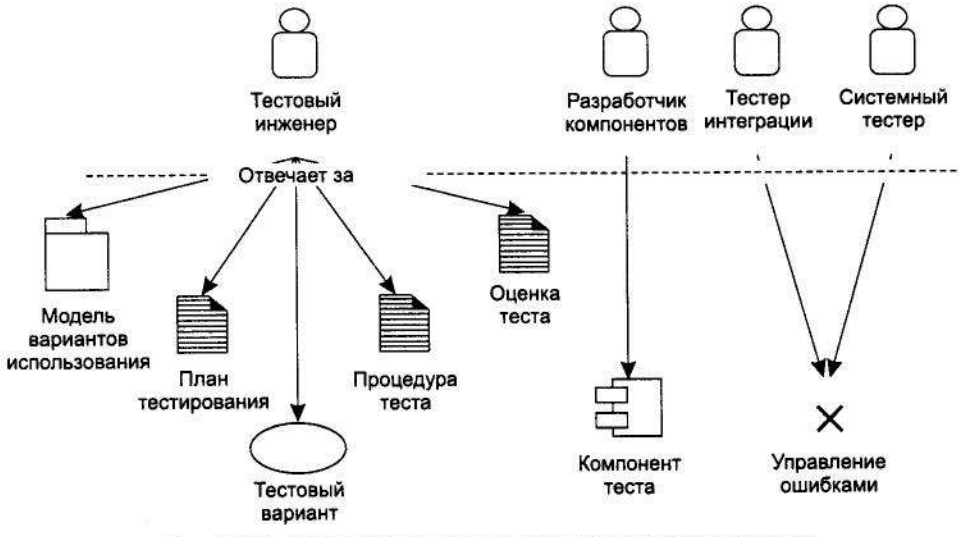


Рис. 9.17. Артефакты и роли для интегрального тестирования

Например, представьте интерфейсы видеоигры *Встреча* (рис. 9.18). Для того чтобы убедиться в корректной работе интерфейсов, мы можем вызывать методы интерфейса последовательно в различных комбинациях. Последовательности, которые мы выбираем, должны выполнять многие комбинации методов интерфейса, но должны также быть осмысленны в контексте игры. В противном случае результат может быть непредсказуемым. Пример понятной последовательности вызовов для видеоигры, результат которых можно наблюдать на мониторе, показан на рис. 9.19.

Последовательность методов, вызванных из `main()` в объекте `InterfaceTest`:

```
1.  //Разместить персонаж игрока в зоне «двор» с характеристиками
    //по умолчанию...
    //Получить объект (интерфейс) EncounterCast
    playerChar = EncounterCast.getThePlayerCharacter();
    //Установить тест зоны «подвал»
    EncounterGame.setLocation( playerChar, «courtyard» );
2. //Установить состояние игры Ожидание
    EncounterGame.setState( «waiting» );
3. //Теперь проверить состояние игры
    state = EncounterGame.getState();
4. //Вывести название состояния
    state.printStateName();
5. //Предположим, что внешний персонаж Фредди в зоне «кухня»
    foreignChar = EncounterCast.getTheForegnCharacter();
    EncounterLayout.setLocation( foreignChar, «kitchen» );
6. //Предположим, внешний персонаж Фредди входит в зону «двор»
```

```
    EncounterLayout.setLocation( foreignChar, «courtyard» );
7. //Проверить, что текущее состояние – Контакт
    state = EncounterGame.getState();
    state.printStateName();
    //Предположим, персонаж игрока убежал в зону «подвал»
    EncounterLayout.setLocation( foreignChar, «dungeon» );
8. //Проверить, изменились ли характеристики игрока
    //должным образом
    ...
```

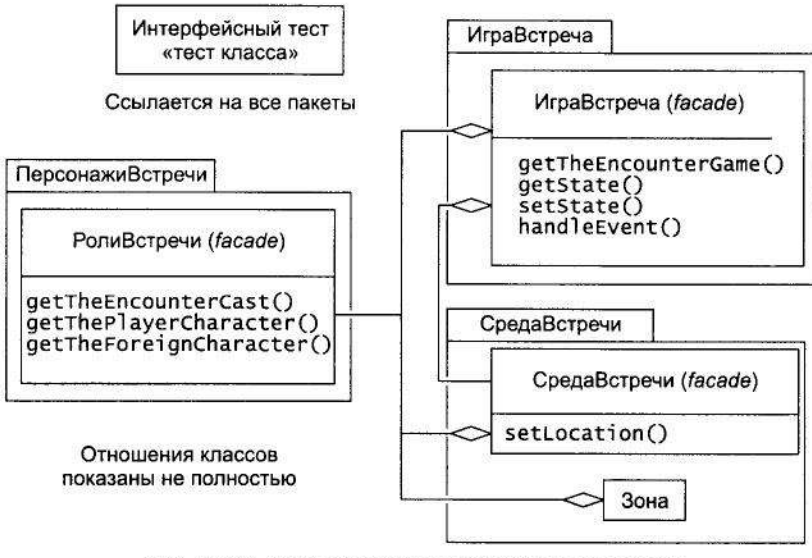


Рис. 9.18. Тестирование интерфейсов для Встречи

Рис. 9.19. Сценарий теста интерфейсов для игры Встреча

9.3.4. Системное тестирование

Системное тестирование является кульминационным моментом интегрального тестирования. Оно состоит из тестов «черного ящика», утверждающих согласованность всей программы с программными требованиями. По мере возможности системные тесты выполняются при запущенной программе в требуемой среде. Иногда, однако, нам приходится довольствоваться лишь запуском системных тестов в среде или конфигурации, отличных от имеющихся у заказчика. Например, мы не будем считать необходимым тестировать апплеты на каждом типе персональных компьютеров. С другой стороны, апплеты следует протестировать на всех основных версиях всех широко распространенных браузеров.

Поскольку системные тесты гарантируют удовлетворение требований, эти тесты должны систематически подтверждать каждое требование. Для принудительной демонстрации выполнения каждого требования потребуется значительный объем тестового кода. На этом этапе мы также должны проверить все варианты использования.

USDP предпринимает попытку организовать большинство требований по вариантам использования. Тестирование в этом случае проще, чем тестирование отдельных атомарных требований.

Основные свойства, которые подвергаются проверке при системном тестировании [74], перечислены ниже.

- Объем.

  Тестируемый продукт исследуется при подаче больших объемов входных данных.

- Удобство и простота использования.

  Требуется измерить реакцию пользователя (например, по шкале 1–10).

- Производительность.

  Измерьте скорость в разных условиях.

- Конфигурация.

  Например, сконфигурируйте систему под разные аппаратные и программные архитектуры и измерьте время настройки.

- Совместимость с другими обозначенными программными приложениями.

  Измерьте время адаптации.

- Надежность и доступность.

  Измерьте период работоспособного состояния в течение длительного времени.

- Безопасность.

  Подвергните систему попыткам несанкционированного доступа. Измерьте среднее время взлома.

- Использование ресурсов.

  Измерьте степень использования оперативной памяти, жесткого диска и т. д.

- Установка.

  Установите программу в разных условиях и измерьте время установки.
- Восстанавливаемость.

  Выполните принудительные действия для зависания программы. Измерьте время восстановления.
- Удобство в эксплуатации.

  Выполните обслуживание программы в разных ситуациях. Измерьте время обслуживания.
- Загрузка и стресс.

  Подвергните программу экстремальному трафику данных и событий.

Надежность и доступность измеряются такими метриками, как *среднее время наработки на отказ* (MTBF – Mean time between failure). Чтобы получить эту величину, сначала нужно сформулировать определение *ошибки* — например, «полное зависание программы». В действительности можно определить несколько уровней ошибок. Для вычисления среднего времени наработки на отказ тестер запускает программу, засекает время, а затем выполняет (в идеале) произвольный сценарий игры до тех пор, пока система не зависнет. Далее отмечается время и вычисляется промежуток времени. Этот процесс выполняется повторно несколько раз. Среднее время наработки на отказ — это среднее арифметическое полученных значений.

Термин *удобство в эксплуатации* относится к простоте или сложности, с которой можно поддерживать работу программы. Например, если экспертное системное приложение работает с собственной базой знаний, то она должна быть легко модифицируема.

9.3.5. Тестирование удобства и простоты использования

Хороший интерфейс может значительно повысить ценность программы. Тестирование *удобства и простоты использования* утверждает приемлемость программы для пользователей.

9.3.5.1. Тестирование требований пользовательских интерфейсов

Основная задача тестирования удобства и простоты использования заключается в гарантии того, что программа удовлетворяет своим требованиям. Вспомните, например, из раздела 3.3.5, что существует огромное количество типов окон, и что они могут появляться несколькими возможными способами. Вдобавок может потребоваться специальная синхронизация. Например, в видеоигре *Встреча* существует определенная задержка между вводом новых значений характеристик персонажа и моментом, начиная с которого новые значения вступают в силу.

Тестирование удобства и простоты использования содержит в себе валидацию этих требований.

Один из способов организации такого тестирования заключается в измерении степени удовлетворенности, полученной пользователями от применения программы.

9.3.5.2. Метрики удобства и простоты использования

Критерии оценки удобства и простоты использования должны быть сформулированы заранее. Например, мы можем потребовать, чтобы произвольная группа из 30 пользователей нашей домашней финансовой программы оценила программу (табл. 9.1). Необходимое количество пользователей определяется статистически и зависит от размеров ожидаемой базы заказчика и желаемой вероятности ошибочного заключения.

Таблица 9.1. Пример значений метрик удобства и простоты использования

Метрика	Среднее значение (из 10)
Простота просмотра	8,5
Простота использования	8,5

На практике данные по удобству и простоте использования будут более подробными, чем указанные в табл. 9.1. Например, Кит [74] перечислил критерии, также важные для тестирования удобства и простоты использования.

- Доступность.

  Насколько легко могут пользователи входить, ориентироваться и выходить из системы?

  Например, измерить среднее затраченное время на...

- Способность реагировать.

  Насколько быстро программа позволяет пользователям достичь определенных целей?

  Например, измерить среднее затраченное время.

- Эффективность.

  Насколько малы необходимые шаги для выбранной функциональности?

  «Малость» устанавливается в теории.

  Например, также измерить минимальное время из пользовательского примера.

- Ясность.

  Насколько легко понимают продукт, пользуются документацией и вызывают справку?

  Например, измерить время, затраченное на стандартный запрос.

Вдобавок к этим характеристикам нам понадобятся метрики, специфичные для конкретной программы, например:

Насколько легко для вас заполнить стандартную форму о случившемся ДТП (по десятибалльной шкале)?

При разработке опросников удобства и простоты использования задача заключается в получении данных, позволяющих инженерам направить усилия на исправление наиболее серьезных упущений, не злоупотребляя временем и терпением пользователей. Накопление данных по удобству и простоте использования может оказаться дорогим процессом, поскольку пользователи часто ожидают компенсации за свое потраченное время и предоставление информации. Например, фирма — клиент автора [74] — разрабатывает программное обеспечение для устройства, используемого врачами. Фирма предоставляет врачам бесплатный ужин и сотни долларов только для того, чтобы они просмотрели и прокомментировали экранные снимки и демонстрации. Разработчик считает, что все эти затраты окупятся.

9.3.6. Регрессионное тестирование

Когда программа разрастается до больших размеров, повышается важность системного тестирования. Это особенно заметно, когда в большую программу вносятся изменения и разработчикам нужно утвердить тот факт, что изменение не повредило существующую функциональность. Первый вопрос, возникающий после интеграции изменения, обычно следующий: «Остался ли продукт тем же, чем он был раньше, с расширенной функциональностью?» Повторное полное инспектирование обычно нерационально, поэтому важным практическим методом получения ответа на этот вопрос является проверка того, что система продолжает проходить тот же определенный набор системных тестов, что и до изменений. Этот процесс верификации называется *регрессионным тестированием.*

Регрессионное тестирование проводится достаточно часто. Если время не позволяет выполнить регрессионное тестирование, выбираются тесты, которые система после внесения изменений с наибольшей вероятностью не пройдет.

9.3.7. Приемосдаточное тестирование

Разрабатывающая программу организация и организация-заказчик являются двумя сторонами, заключившими контракт. После завершения работ мудрый разработчик получает окончательное утверждение заказчика, согласно которому можно начинать поставку программы. *Приемосдаточные тесты* разрабатываются для убеждения клиента в том, что указанная программа действительно создана. Приемосдаточные тесты могут ничем не отличаться от системных тестов, созданных разработчиком, но на этот раз они должны быть официально засвидетельствованы организацией-заказчиком и проверены на целевых платформах.

От заказчиков часто требуется промежуточная частичная оплата на основе промежуточных версий. Это частичные реализации и проекты, которые также требуют проведения приемосдаточного тестирования.

9.3.8. Тестирование инсталляции

Тот факт, что мы протестировали программу в нашей собственной среде, вовсе не означает, что программа будет корректно работать в среде заказчика, поскольку при переходе из одной среды в другую открываются необъятные просторы для новых ошибок. *Тестирование инсталляции* состоит из тестирования программы в целевой аппаратной конфигурации. Это влечет за собой инсталляцию программы в целевой среде и выполнение комплекта системных тестов. Для заархивированных прикладных программ тестирование инсталляции состоит из выполнения программы на платформах, представляющих собой типовые среды заказчиков.

9.4. Документирование интеграции и тестирования

9.4.1. Стандарты документации по тестированию

Стандарт ANSI/IEEE 829-1983 на Документацию по тестированию программного обеспечения (STD — Software Test Documentation), заново подтвержденный в 1991 году, состоит из следующих разделов:

1. **Введение**
2. **План тестирования**

   Тестируемые элементы, границы, подход, ресурсы, расписание, персонал.
3. **Проект тестирования**

   Тестируемые элементы, подход, план в подробностях.
4. **Тестовые варианты**

   Наборы входных данных и событий.
5. **Тестовые процедуры**

   Шаги настройки и выполнения тестовых вариантов.
6. **Отчет о проведении тестирования элементов**

   Тестируемый элемент, физическое местоположение результатов, ответственный за проведение тестов.
7. **Журнал испытаний**

   Хронологическая запись, физическое местоположение теста, название теста.
8. **Отчет о происшествиях во время тестирования**

   Документирование любого события, имевшего место во время тестирования, требующего дальнейших расследований.
9. **Итоговый отчет о тестировании**

   Итог всего вышеперечисленного.

1. В разделе *Введение* объясняется содержание тестов и их общие принципы. Например, если программа управляет оборудованием помещения скорой помощи, именно в этом разделе мы должны объяснить наш общий подход к тестированию моделей, сводящийся к тестированию в условиях данных помещений.
2. *План тестирования* объясняет, как следует организовать персонал, программы и оборудование, чтобы выполнить тестирование. Например: «Временной модуль будет тестировать Джо на протяжении недель 30–33; модуль мониторинга сердца будет тестировать Сьюзан на протяжении недель 26–30; интеграцию этих двух модулей будет тестировать Эд на протяжении недель 31–33; ... ».
3. *Проект тестирования* отражает следующий уровень детализации после плана тестирования. Он раскрывает значение соответствующих программных элементов, описывает порядок, в котором их следует тестировать, называет тестовые варианты, которые следует применить. Например: «Джо будет тестировать временной модуль отдельно на протяжении недель 30–33, используя тестовую процедуру 892 и драйвер 8910; Сьюзан будет тестировать модуль мониторинга сердца отдельно на протяжении недель 26–30, используя тестовую процедуру 555 и драйвер 3024; Эд будет тестировать сборку, интегрирующую эти два модуля (сборка 7), используя...».
4. *Тестовые варианты* состоят из наборов входных данных и точных входных сигналов, которые должны использоваться для выполнения теста. Например, модуль мониторинга сердца должен работать по тестовому файлу 892, в котором содержатся конкретные данные по конкретному пациенту в конкретное время. Мы должны точно указать, где находится этот тестовый файл.
5. *Тестовые процедуры* — это полные подробные шаги выполнения плана тестирования. Сюда входят все процедуры настройки, имена необходимых файлов с исходными данными и объектным кодом, выходные файлы, log-файлы, файлы с тестовыми вариантами и отчеты. Например, тестовая процедура 892 могла бы быть аналогична приведенной ниже:
   1) откомпилировать временной модуль вместе с драйвером 8910;
   2) установить путь ...;
   3) загрузить содержимое файла 672 в файл, названный `input`, в том же каталоге, что и объектный код;
   4) выполнить код со следующими параметрами...;
   5) в появившемся окне ввести `Джонс` в текстовом поле имени...;

   ...

   Причиной использования такого уровня детализации является тот факт, что когда тест выявляет дефект, важно знать точные обстоятельства, при которых этот дефект возник. Без подробной пошаговой документации тестирования такие тесты невозможно достоверно воспроизвести, и дефект, возможно, вообще не удастся выявить повторно. В этом случае ошибку исправить трудно, если не невозможно.

6. *Отчет о проведении тестирования элементов* резюмирует запускаемые нами тесты, список ответственных лиц, используемые версии продукта и т. д.
7. *Журнал испытаний* представляет собой подробный текущий отчет о полученной во время тестов информации. Он может оказаться полезен при попытке воспроизвести ситуации, в которых тест завершился неудачно.
8. *Отчет о происшествиях* уточняет заслуживающие внимание события, происшедшие во время тестирования. Примерами могут быть отклонения от нормальной работы программы и допущенные в процессе тестирования ошибки.

Формат IEEE можно использовать для тестов большинства типов, упомянутых в этой главе. Этот формат использован в примере с видеоигрой *Встреча* в конце главы для построения интегральных тестов.

9.4.2. Организация документации по интеграции и тестированию

Полезным документом, в котором описан процесс сборки частей программы, является документ управления конфигурациями (в терминах IEEE это SCMP) (рис. 9.20). Организация этого документа показывает, что описание SCMP содержится в SPMP, а также показывает список сотрудников, отвечающих за SCMP. Сам SCMP описывает конкретные процедуры для поддержки (хранения, маркирования, согласования и т. д.) различных версий различных документов, в том числе и SPMP. В нем также точно определено местоположение этих документов. Последняя спецификация разрастается, и ее предпочтительнее описать в приложении к SCMP. SCMP и его приложение должны ссылаться на документацию по тестированию (в терминах IEEE – STD) для четкого отслеживания выполняемых тестов, соответствующих тестовых вариантов, процедур, планов и т. д. и существующих версий кода, которые тестируются.

Связь между различными тестовыми документами и их связь с существующей документацией показана на рис. 9.21. STD-документация по тестированию содержит в себе все типы тестирования, описанные в этой главе. Она может также содержать модульное тестирование (в зависимости от степени документирования модульного тестирования). Стандарт IEEE для документации по тестированию приводится ранее в начале раздела 9.4.1; эта организация документации применяется к каждой сборке и к каждому из разнообразных типов тестов (рис. 9.21).

Артефакты различных тестов используются вторично, что показано на рис. 9.21 пунктирными линиями. Например, при тестировании *сборки* обычно используются планы тестирования, проекты, варианты и процедуры, разработанные для тестирования предыдущих сборок. *Системное* тестирование использует артефакты (тестовые варианты и т. д.), которые были разработаны для тести-

рования финальной сборки; *приемосдаточные* тесты используют артефакты *системного* тестирования, а тестирование *инсталляции* использует артефакты *приемосдаточного* тестирования.

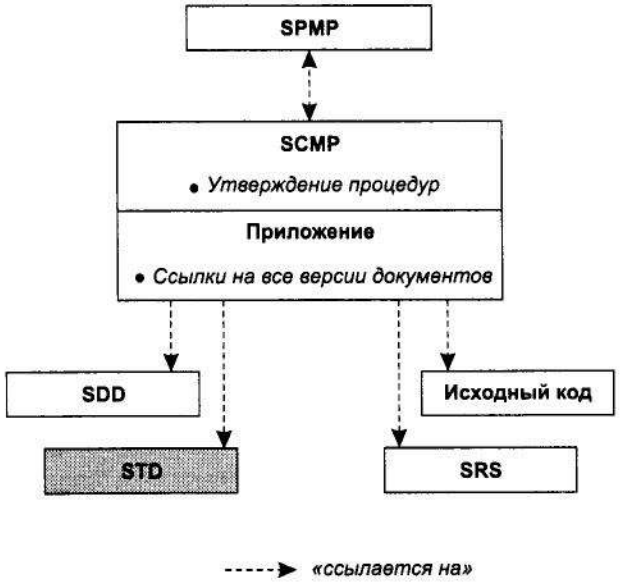


Рис. 9.20. STD в контексте разработки

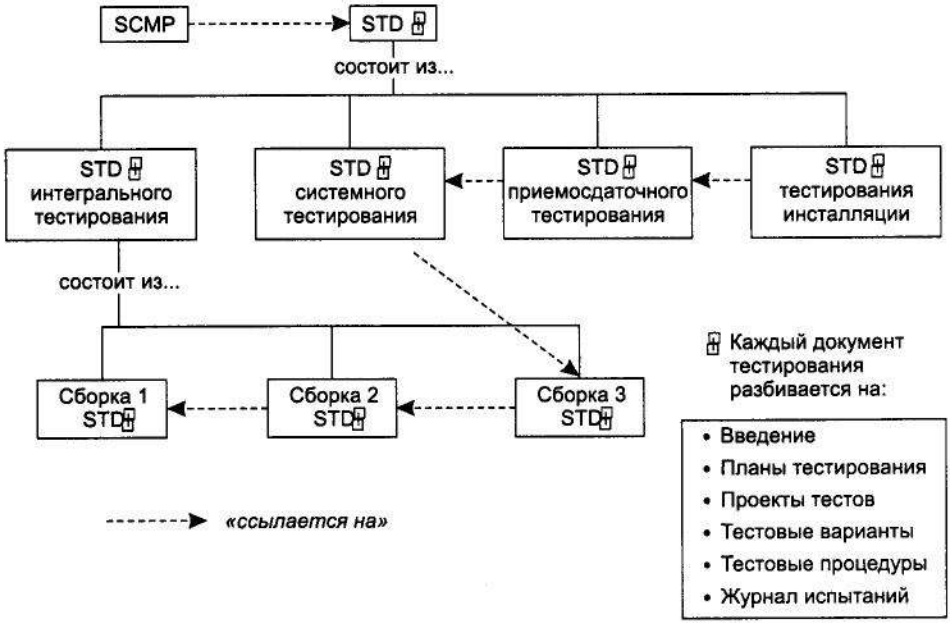


Рис. 9.21. Организация документации интегрального и системного тестирования

9.5. Итерации перехода

После интеграции программного приложения необходимо выполнить некоторые действия, прежде чем выпускать его новую версию. Требуемые действия объединены Якобсоном в итерациях *перехода* (согласно USDP). Цели этих итераций представлены на рис. 9.22. На этом же рисунке подведен итог относительного количества требований, анализа и т. д., необходимых для этих итераций на стадии перехода.

Рис. 9.22. Цели итераций перехода

9.5.1. Альфа- и бета-версии

Во многих случаях предполагаемые пользователи хотят наряду с заказчиками участвовать в процессе системного тестирования. Этот процесс управляется с помощью альфа- и бета-версий.

- Альфа-версия:
  - внутренние и высоконадежные пользователи;
  - множественное тестирование;
  - позволяет получить предварительную оценку реакции пользователя;
  - выгодна разработчикам третьей стороны;
  - препятствует конкуренции.
- Бета-версия:
  - избранные заказчики;
  - *множественное тестирование;*
  - позволяет получить реакцию заказчика.

Альфа-версии даются внутренним пользователям или строго отобранной надежной группе внешних пользователей для раннего предвыпускного использования.

Назначение альфа-версий — предоставить организации-разработчику обратную связь и информацию о дефектах от группы людей, превосходящей тестеров в количестве, без изменения репутации пока не выпущенного продукта. После распространения альфа-версии выпускается бета-версия.

Бета-версии раздаются части сообщества заказчиков с учетом того, что заказчики должны будут докладывать об обнаруженных ошибках. Кроме того, альфа- и бета-версии используются для убеждения потенциальных клиентов в том, что помимо обещаний разработчика существует уже почти готовый продукт. Иногда распространение предварительного выпуска продукта является стратегическим приемом, используемым для внушения покупателям мысли о том, что они не должны покупать конкурирующие продукты, а вместо этого следует подождать выхода программы, которая проходит стадию бета-тестирования. Сопутствующие этические вопросы не освещаются в этой книге.

Основной мотивацией альфа- и бета-тестирования является получение более полной информации о продукте. Разработчики могут получить информацию о программе (обычно о ее программном интерфейсе), чтобы в будущем иметь возможность начинать разработку программ, использующих эту. Пользователи получают возможность обдумать покупку этой программы.

9.5.2. План итераций перехода

Основные этапы итераций перехода (финальных) показаны на рис. 9.23.

Рис. 9.23. План итераций перехода

Критерием остановки являются условия, при которых продукт следует допустить к приемосдаточному тестированию. Если эти условия не определены заранее, тестирование обычно проводят до тех пор, пока не закончится время: обычно это не самый эффективный способ использования времени тестирования. Примером критерия остановки может быть: «Максимум две ошибки среднего или низкого уровня найдено за неделю бета-тестирования». Приведем классификацию критериев остановки, предложенную в [74].

- Завершение конкретной методологии теста.

  Завершить процедуры метода или инструмента.

- Оценка границ в процентах по каждой категории.

  Предопределить процент каждой и показать, как его рассчитывать.

  Например, «95 % утверждений».
- Скорость нахождения ошибок.

  Предопределить скорость при заданном уровне важности.

  Например, «2 дефекта средней важности или менее на каждые 100 часов работы».
- Общее число обнаруженных ошибок.

  Вычислить (если возможно) из процента оставшихся ошибок.

  Предопределить процент обнаруженных ошибок.

  Например, «95 % оценки существующих ошибок найдено».

На рис. 9.24 упоминаются *оставшиеся ошибки*, но как мы можем оценить число оставшихся ошибок? Один из способов называется «засев». Он состоит из добавления некоторого количества ошибок в программу и определения их процентного соотношения среди ошибок, найденных независимым тестером за определенный срок. Это число затем используется для оценки числа оставшихся дефектов.

Например, если 3 из 50 посеянных ошибок найдены за период тестирования, можно оценить количество необнаруженных ошибок на каждую обнаруженную как 47/3 = 15,67. Таким образом, если за тот же период было найдено 100 непосеянных ошибок, в системе осталось порядка 100×15,67 = 1567 необнаруженных ошибок.

При использовании некоторых из этих критериев в проектах могут быть использованы графики для определения сроков выпуска продукта (рис. 9.24). В этом примере использовались три критерия остановки. Критерий остановки для скорости обнаружения ошибок гласит: «Не более семи ошибок найдено за каждые 1000 часов тестирования одной недели на протяжении хотя бы четырех последовательных недель». (Чтобы уместить 1000 часов тестирования в одну неделю, тестирование следует проводить параллельно на нескольких копиях одной программы.) Это банковская программа, и тестирование операций *депозита* и *снятия* проводится отдельно. В случае, изображенном на рисунке, последним критерием, который необходимо выполнить, является «Процент протестированных операций снятия», поэтому продукт не будет выпущен до устранения этого дефекта (неделя 26).

9.6. Качество в интеграции, верификации и валидации

9.6.1. Качество, к которому следует стремиться

Хорошие планы интеграции всегда тщательно продуманы, а эффективные системные и интегральные тесты подробны и всесторонни. Приведенные ниже метрики содействуют этим свойствам.

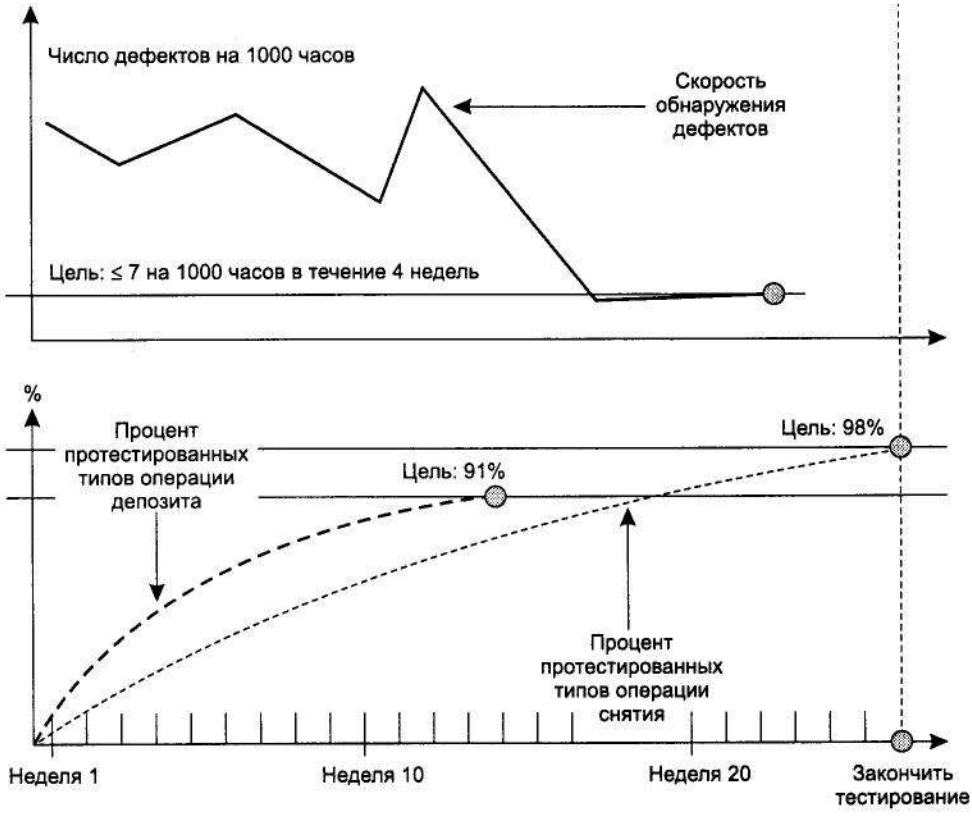


Рис. 9.24. Критерий остановки: графическое представление

9.6.2. Метрики интегрального и системного тестирования

Ниже приведены метрики, взятые из IEEE 982.1-1998 — словаря стандартных метрик [59].

- IEEE 1. Плотность отказов = [Число уникальных отказов, найденных при тестировании] / [Число строк кода].
- IEEE 2. Плотность дефектов = [Число уникальных дефектов, найденных при тестировании] / [Число строк кода].
- IEEE 5. Функциональный охват теста = [Число протестированных функциональных требований] / [Общее число требований].
- IEEE 10. Индекс зрелости программы = [$M - F_a - F_c - F_d$] / M, где
  - M — число частей имеющегося базиса;
  - F_a — число добавленных частей в текущем базисе по сравнению с предыдущим базисом;

- F_c — число частей в текущем базисе, измененных по сравнению с предыдущим базисом;
- F_d — число частей, удаленных из предыдущего базиса.

«Частями» могут быть функции, классы, пакеты, модули и т. д. Развитые программы имеют индекс зрелости, близкий к единице. Это означает, что число затронутых частей невелико по сравнению с общим числом компонентов.

- IEEE 18. Надежность работы выражается вероятностью того, что в k произвольных случаях работы программа вернет корректный результат. Эта величина оценивается через выполнение некоторого числа запусков программы (N) и вычисления числа случаев успешной работы (S). Вероятность успеха, таким образом, вычисляется как S/N, а вероятность возможности отработать k раз успешно — как произведение вероятностей каждого успешного запуска, то есть $[S/N] \times [S/N] \times ... \times [S/N]$, или $[S/N]k$. Входные данные для каждого случая выбираются произвольно и независимо от предыдущего запуска.
- IEEE 20. Среднее время обнаружения k отказов. Это значение вычисляется аналогично надежности работы (см. IEEE 18 выше).
- IEEE 21. Уровень безупречности программы. Эта метрика оценивает отсутствие отказов в программе на операционной стадии [59].
- IEEE 22. Оценка числа оставшихся отказов (методом засева). Эта оценка получена путем «засеивания» в программу N произвольных отказов. Если s — число найденных засеянных отказов, а f — число других отказов, найденных за тот же период тестирования, оценка равна $f \times N / s$.
- IEEE 24. Охват теста. Это число оценивает законченность выполненного тестирования (например, доля выполненной работы, умноженная на долю выполненных тестов). Вот формула для вычисления этой оценки:

  ОТ (в процентах) = [[Число реализованных требований] / [Число требований]] × [[Число протестированных программных примитивов] / [Общее число примитивов в программе]] × 100.

  Программные примитивы — это тестируемые модули программы. Сюда относятся методы, классы и пакеты.
- IEEE 30. Средний период ошибки (MTTF — Mean-time-to-failure). Измеряется посредством запоминания промежутков времени между всеми парами замеченных последовательных ошибок и их усреднения. При измерении промежутков обычно используется фактическое истекшее время, а не время центрального процессора.
- IEEE 36. Точность теста. Этот тест оценивает надежность процесса тестирования и представляет собой побочный продукт описанного выше теста 22.

  Точность теста = N_f/N, где N — это число засеянных отказов, а N_f — это число засеянных ошибок, найденных во время тестирования.

9.6.3. Инспектирование системного и интегрального тестирования

Несколько аспектов интеграции поддаются процессу инспектирования. Сюда относятся части SCMP, относящиеся к последовательности интеграции, и различные планы тестирования, например План интеграции (IP) и План тестирования программного обеспечения (STP). Примером дефекта в плане интеграции может быть отсутствие необходимого элемента (модуля, класса, метода и т. д.) на некоторой стадии интеграции, например, когда необходимо, чтобы элемент присутствовал для формирования тестируемой сборки или варианта использования. Примером дефекта в интегральном тесте является отсутствие тестового шага, являющегося частью соответствующего варианта использования.

Последовательность сборок и их тестов может быть очень сложной. Отсюда и вытекает выгода от инспектирования.

9.6.4. Привлечение группы контроля качества к интегральному и системному тестированию

Сотрудники группы контроля качества обычно более активно задействованы на этапах тестирования сборок и системного тестирования, чем на любом другом этапе процесса.

Для понимания важности и сложности процесса контроля качества рассмотрим пример разработки программы для составления прогноза погоды. Одна из основных функций этой программы заключается в преобразовании огромных объемов данных в графические изображения. Нужно учесть, что данные постоянно меняются, а также существует множество способов представления информации. Конкуренция и частые запросы служб прогноза об улучшении характеристик программы приводят к постоянным изменениям и улучшениям программы. Вдобавок производитель стремится выпускать новые, улучшенные версии для получения большей прибыли. В организации-разработчике такого типа обычно имеется группа контроля качества, тестирующая новые характеристики и проводящая регрессионное тестирование. Представьте себе перспективы производителя, если система изображения откажет на глазах у миллионов людей.

Одной важной проблемой группы контроля качества в этом случае является воспроизводимость. Группа контроля качества должна воспроизвести сеанс работы, в котором пользователь щелкнул мышью на каком-то поле в некоторый момент изображения грозы. Инструменты, записывающие и воспроизводящие действия пользователя, описываются далее в разделе 9.7. Они могут помочь, но не охватывают абсолютно все типы программ. Существует значительный объем дополнительной работы, которую необходимо выполнить для измерения и поддержания качества.

При обнаружении ошибок на системном уровне необходимо оповестить соответствующих сотрудников. Это может оказаться важным дипломатическим, управленческим и техническим заданием. Дипломатическим, потому что разработчики не любят слышать о своих ошибках; управленческим, потому что согла-

сованные дефекты необходимо проследить, и ответственность за это ляжет на многих людей; техническим, так как определение причин ошибки может быть очень сложным.

9.6.5. Системная интеграция и модель CMM

Время окончания проекта является удачным моментом для оценки использованного процесса и для организации улучшений процесса. Типичная организация стремится перейти на следующий уровень CMM.

1. Начальный.

   Неопределенный процесс, построенный конкретно для текущей цели.

2. Повторяемый.

   Отслеживаются фактические стоимость, план, функциональность.

3. Установленный.

   Документированный, стандартизированный настраиваемый процесс.

4. Управляемый.

   Подробные оценки, управление.

5. Оптимизированный.

   Последовательные дискретные улучшения процесса.

В качестве примера представьте себе, что наша организация находится на уровне 3 и пытается достичь уровня 4. Таким образом, команде придется тщательно измерять и контролировать проект (а не позволять проекту управлять группой разработчиков). Подведение итогов работы может иметь форму, показанную в табл. 9.2.

Таблица 9.2. Пример подведения итогов: анализ требований через системную интеграцию

	Набор метрик	Управляемость	Действия
Требования	Поддерживаются только 2 из 4 метрик требований	Хорошая — проблемой пренебрегли, времени хватает	Поручить инженеру группы контроля качества поддержку SRS и всех четырех метрик. Л. Д. к 01.03
Проект	Не смогли определить метрики	Низкая — справились за 140 % запланированного времени	Х. Р. должен выбрать три лучшие метрики для программы такого типа; оценить стоимость и график выполнения; к 05.03. Л. Д. должен описать, как нам следует расставить приоритеты в работе по проекту; к 01.04. С. Т. должен решить, достаточно ли времени было выделено на разработку проекта или процесс был неэффективен (и сформулировать причины этого); к 15.03

	Набор метрик	Управляемость	Действия
Реализация	Хорошая	Выполнена за 130 % запланированного времени	Просмотреть методы оценки числа строк кода. Определить три главные причины превышения. Д. А. к 05.03
Интеграция и выпуск версии	Использовалось слишком много бесполезных метрик	Адекватна	Удалить метрику «скорость подписи бета-узла». Заново определить эффективность других использовавшихся метрик. Б. В. к 10.03

9.7. Инструментальные средства интегрального и системного тестирования

Для абсолютно полного тестирования обычно требуются автоматизированные инструменты тестирования. Якобсон и другие [64] предложили в лучшем случае автоматизировать по крайней мере 75 % тестов, а остальную часть тестов проводить вручную. Некоторые возможности инструментов тестирования перечислены ниже.

1. **Запись и воспроизведение.** Без возможности записывать и воспроизводить события мыши и клавиатуры качество тестирования падает, так как тестерам приходится выполнять это вручную. Это утомительно и дорого. Вдобавок результаты могут не быть в точности сравнимыми, поскольку люди не могут абсолютно точно повторять действия. Классификация инструментов записи и воспроизведения [74] показана ниже.
   - Собственные / встроенные в программу.

     Тестовые приложения, интегрированные в тестируемую программу.

     Могут скомпрометировать тестируемую программу.

     Наименее дорогие.
   - Собственные / встроенные аппаратно.

     Тестовая аппаратура, интегрированная с тестируемой программой.

     Может скомпрометировать тестируемую программу.
   - Невстроенные.

     Используют отдельную тестовую аппаратуру.

     Не компрометируют тестируемую программу.

     Наиболее дорогие.

   Большинство распространенных инструментов записи-воспроизведения являются тестами, встроенными в программу. Примером невстроенных системных тестов являются тесты военной оперативной системы управления реального времени, в которой взаимодействие с пользователем эмулируется с помощью отдельных устройств, подключенных к тестируемой программе. Внешние устройства программируются так, чтобы они выдавали входные сигналы и программа не могла отличить эти сигналы от ввода данных реальным пользователем.

Инструменты записи-воспроизведения могут оказаться очень полезными, но они крайне чувствительны к изменениям в пользовательском интерфейсе. Небольшое изменение пользовательского интерфейса может свести на нет весь набор автоматически выполняемых тестов.

2. **Неоднократный запуск тестовых сценариев.** Возможность автоматически выполнять тесты программы дает тестерам возможность не повторять один и тот же тест вручную с различными параметрами.
3. **Запись результатов тестов.** Это освобождает тестеров от необходимости реализовывать эту функцию.
4. **Учет времени.** Автоматические тестовые инструменты могут измерять и записывать истекшее время и загрузку центрального процессора.
5. **Запись ошибок в работе программы.** Некоторые автоматические инструменты тестирования могут записывать ошибки, возникшие в ходе работы программы.
6. **Управление регрессионным тестированием.** Вспомните, что регрессионное тестирование необходимо для утверждения того факта, что изменения предыдущей версии не добавили новых ошибок. Регрессионные тесты меняются во времени по мере реализации все больших возможностей. Некоторые автоматические инструменты тестирования могут вести учет этих тестов и применять их по требованию.
7. **Генерация тестовых отчетов.** К автоматическим инструментам тестирования относятся генераторы тестовых отчетов, исключающие необходимость писать многочисленные отчеты о тестировании вручную либо создавать свой собственный инструмент генерации отчетов. Последнее было частично сделано для нашего примера в конце главы.
8. **Генерация тестовых данных.** Среди наиболее полезных инструментов тестирования есть инструменты для генерации тестовых данных. Эти инструменты генерируют входные данные, удовлетворяющие большинству тестов «черного» и «белого ящика», обсуждавшихся в этой главе и в главе 8. Примером является генерация произвольных комбинаций входных данных. Некоторые тестовые инструменты также облегчают тестирование «серого ящика» и системные тесты, выполняющие проверку взаимодействия модулей. Не следует ожидать, что эти инструменты будут генерировать корректные выходные данные для каждого тестового варианта, поскольку эта возможность является целью создаваемой нами программы!
9. **Тестовые инструменты использования памяти.** Эти инструменты очень полезны при тестировании работы программ. Одни из этих инструментов просто сообщают статистику в форме таблиц или графиков, в то время как другие могут обнаружить некоторые ошибки. Их потенциальные возможности таковы.

   - Утечка памяти.

     Определить растущие объемы неиспользуемой памяти, неумышленно возникшие в результате реализации.

- Поведение использования памяти.

  Подтвердить ожидания.

  Определить помехи.
- Поведение границ данных.

  Например, подтвердить целостность массивов или определить достижение граничных значений.
- Инициализация переменных.

  Указать неинициализированные переменные
- Перезапись активной памяти.

10. **Инструменты управления тестовыми вариантами.** Тестовые инструменты управления обычно обладают следующими возможностями [74].
    - Предоставить пользовательский интерфейс.

      Для управления тестами.
    - Организовать тесты.

      Для простоты использования.

      Для сопровождения.
    - Управлять сессиями выполнения тестов.

      Для выполнения выбранных пользователем тестов.
    - Интегрировать с другими тестовыми инструментами.

      Для записи и воспроизведения.

      Для анализа области охвата.
    - Предоставить отчеты.
    - Предоставить документацию.
11. **Анализ охвата.** Анализаторы охвата получают на вход продукт вместе с тестовым набором. Они предоставляют анализ охвата набора тестов. Эти анализаторы могут также проверить различные типы охвата тестирования, в том числе и охват утверждений.

Тестирование требует повторяющегося использования форм. Шаблоны документов являются наиболее простыми, но наиболее широко используемыми «инструментами» тестирования. Шаблоны могут основываться на стандартах документации по тестированию, например ANSI/IEEE 829-1983 STD (подтвержденных в 1991 году).

Хотя программы автоматического тестирования способны выполнить многие задачи тестового программирования, их использование часто требует значительных программистских навыков. Например, инструменты записи действий мыши и клавиатуры должны отслеживать соответствующие события, а это требует работы программистов с хорошим знанием процесса генерации событий, так чтобы тестовые инструменты могли перехватить работу программы должным образом.

9.8. Подведение итогов

В этой главе описана фаза интеграции в разработке программы, состоящая из планирования, выполнения и инспектирования сборок. Резюмируем различные формы тестирования, рассмотренные в этой главе.

- Процесс интеграции: выполнен по аккуратно спланированным сборкам.
- Тестирование интеграции: каждая сборка.
- Системное тестирование: программа целиком.
- Регрессионное тестирование: проверить, чтобы изменения не компрометировали возможности, существовавшие до внесения изменений.

Упражнения

Ответы и подсказки для упражнений, помеченных символами «о» или «п», приводятся в конце этой главы.

Вопросы для проверки

П9.1°. При переходе от одной фазы к следующей часто происходит потеря информации. Назовите от трех до пяти переходов, при которых часто наблюдаются такие потери.

П9.2°. В чем разница между верификацией и валидацией?

П9.3°. Назовите от четырех до восьми различных видов тестов и укажите, для чего они необходимы.

П9.4°. Что такое *сборки* и как они связаны с итерациями?

П9.5°. Назовите от трех до пяти артефактов, за которые несут ответственность тестовые инженеры.

П9.6°. Что такое регрессионное тестирование и почему оно необходимо?

П9.7°. В чем разница между альфа- и бета-тестированием?

П9.8°. Что такое приемосдаточное тестирование?

П9.9°. Назовите от четырех до одиннадцати метрик для интегрального и системного тестирования.

Упражнения в команде

К9.1. («Интеграция») Получите спецификации проекта от двух других команд в классе. Неформально определите новую программу, содержащую важные элементы этих приложений. Определите план интеграции для сборки этого нового программного приложения.

Критерии оценки.

1. Степень ясности плана («Отлично» — очень ясный, понятный план).
2. Степень, в которой план содержит подходящий порядок действий («Отлично» — исполнимый и уместный порядок операций).

Общие упражнения

О9.1. На рис. 9.25 показана архитектура программы, эмулирующей обслуживание клиентов банка. Предоставьте план сборки этой программы.

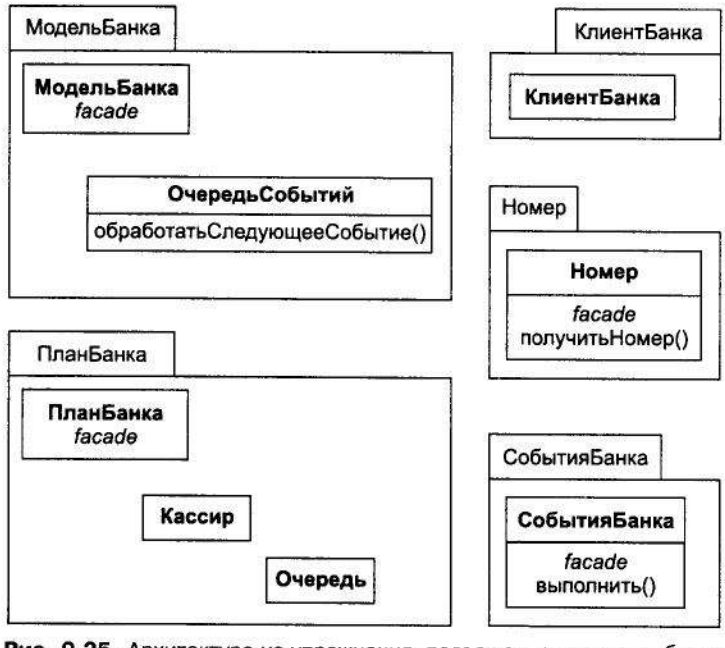


Рис. 9.25. Архитектура из упражнения, посвященного модели банка

Критерии оценки.

1. Степень ясности плана («Отлично» — очень ясный, понятный план).
2. Степень, в которой план содержит подходящий порядок действий («Отлично» — исполнимый и уместный порядок операций).

О9.2. Опишите разные уровни тестирования, которым вы бы подвергли программу из предыдущего упражнения, и укажите, какие из классов, показанных на рис. 9.25, будут участвовать в каждом из этих тестирований. (В создании полной программы будет участвовать гораздо больше классов, но вам не обязательно указывать их все.)

Ответы

П9.1. Потери информации обычно происходят при преобразовании:

- ✦ требований в архитектуру;
- ✦ архитектуры в детальный проект;
- ✦ архитектуры в спецификации интерфейсов;

- ✦ детального проекта в код отдельных функций;
- ✦ спецификации интерфейса в код модулей.

П9.2. Верификация проверяет, правильно ли мы строим программу. Валидация проверяет, правильную ли программу мы строим.

П9.3. Тесты *функций, классов и модулей* являются модульными тестами, проверяющими эти физические единицы.

Тесты *интерфейсов* валидируют способ, которым модули обмениваются информацией друг с другом.

Интегральные тесты валидируют сборки частичной программы.

Системные тесты валидируют работу программы в целом.

Приемосдаточные тесты подтверждают, что программа действительно делает то, что обещали ее разработчики.

Тесты *инсталляции* подтверждают, что программа работает согласно спецификации в запланированных физических средах.

П9.4. Каждая итерация состоит из последовательности сборок. Каждая сборка — это реализация части программы, разработанная для удобства процесса сборки. Каждая сборка использует в качестве базиса предыдущую сборку.

П9.5. Тестовые варианты, процедуры, планы, оценки и, возможно, модели вариантов использования.

П9.6. Регрессионные тесты разрабатываются для утверждения того факта, что изменение или добавление в коде не испортило имевшиеся раньше возможности. Такие тесты необходимы, поскольку изменения в коде могут полностью изменить поведение программы. Изменения в существующем поведении могут быть результатом дефективных изменений или дефективного существующего проектирования (кода).

П9.7. Альфа-версии даются внутренним пользователям или жестко отобранным представителям заказчика. Бета-версии предоставляются широкому сообществу пользователей.

П9.8. Приемосдаточное тестирование является официальным процессом тестирования, с помощью которого клиент может удостовериться, что продукт отвечает требованиям контракта.

П9.9. Ответ на этот вопрос можно найти в разделе 9.6.2.

Пример 1. План управления конфигурациями программного обеспечения (SCMP). Приложение A: План создания базиса интеграции

[Примечание для студентов. Нам необходимо описать порядок, в котором программа будет интегрироваться. SCMP является подходящим местом для этого описания, поскольку он описывает конфигурации итераций и сборок.]

История версий этого документа:
1 ноября 1998. Э. Брауде: начальный черновик.
4 апреля 1999. Э. Брауде: внесены исправления.
23 августа 1999. Р. Боствик: просмотрены документы, даны рекомендации.
24 августа 1999. Э. Брауде: рекомендации выполнены.
26 августа 1999. Э. Брауде: просмотрено, расширены комментарии.

1. Введение

Во время процесса интеграции программа *Встреча* конструируется по стадиям или сборкам. Это приложение описывает конфигурацию первых трех сборок. Интегральное тестирование основывается на этих сборках. Последняя сборка будет базисом для системного тестирования.

2. Создание базиса интеграции

Три последовательные сборки первой версии видеоигры *Встреча* показаны на рис. 9.26. Первая сборка состоит из пакетов *ПерсонажиИгры* и *ПерсонажиВстречи*. Вторая сборка использует первую. Она состоит из пакета *СредаВстречи*, соответствующего ему каркаса и первой сборки. Третья сборка использует первые две. Она состоит из пакета *ИграВстреча*, соответствующего каркаса, а также сборок 1 и 2.

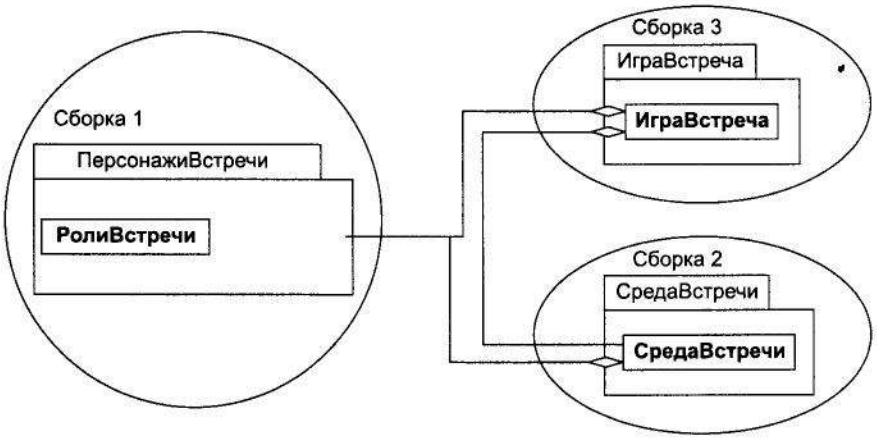


Рис. 9.26. План интеграции видеоигры Встреча

2.1. Интеграционная сборка 1

Сборка 1 показана на рис. 9.27. Сборка 1 реализует каркасные пакеты *ПерсонажиИгры* и *ПерсонажиВстречи*.

2.2. Интеграционная сборка 2

Сборка 2 показана на рис. 9.28. Она состоит из пакета *СредаВстречи* и каркасного пакета *ИгроваяСреда*, а также составляющих первой сборки. Пакетами

ИгроваяСреда и *СредаВстречи* используются соответственно классы *ПерсонажИгры* и *ПерсонажВстречи* из первой сборки. Двор, подвал и гостиная являются примерами зон. Некоторые из этих зон соединены. Например, существует соединение между гардеробом и двором.

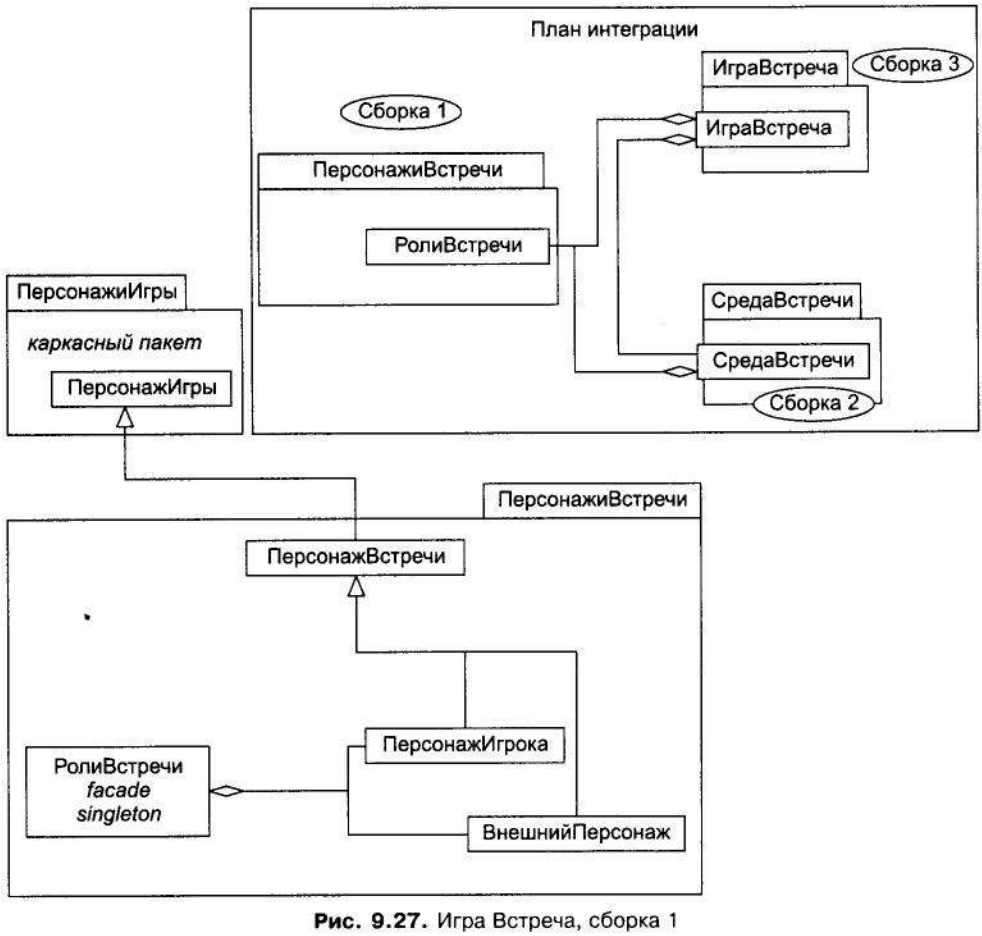


Рис. 9.27. Игра Встреча, сборка 1

Декомпозиция программы, существующая на данный момент, на элементы каркаса показана на рис. 9.29. Пакет *ИграВстреча* и его каркасный пакет *РолеваяИгра* отсутствуют на рисунке, поскольку они являются частями третьей сборки.

2.3. Интеграционная сборка 3

Финальная, третья сборка показана на рис. 9.30. Сборка 3 состоит из пакета *ИграВстреча*, каркасного пакета *РолеваяИгра*, сборок 1 и 2.

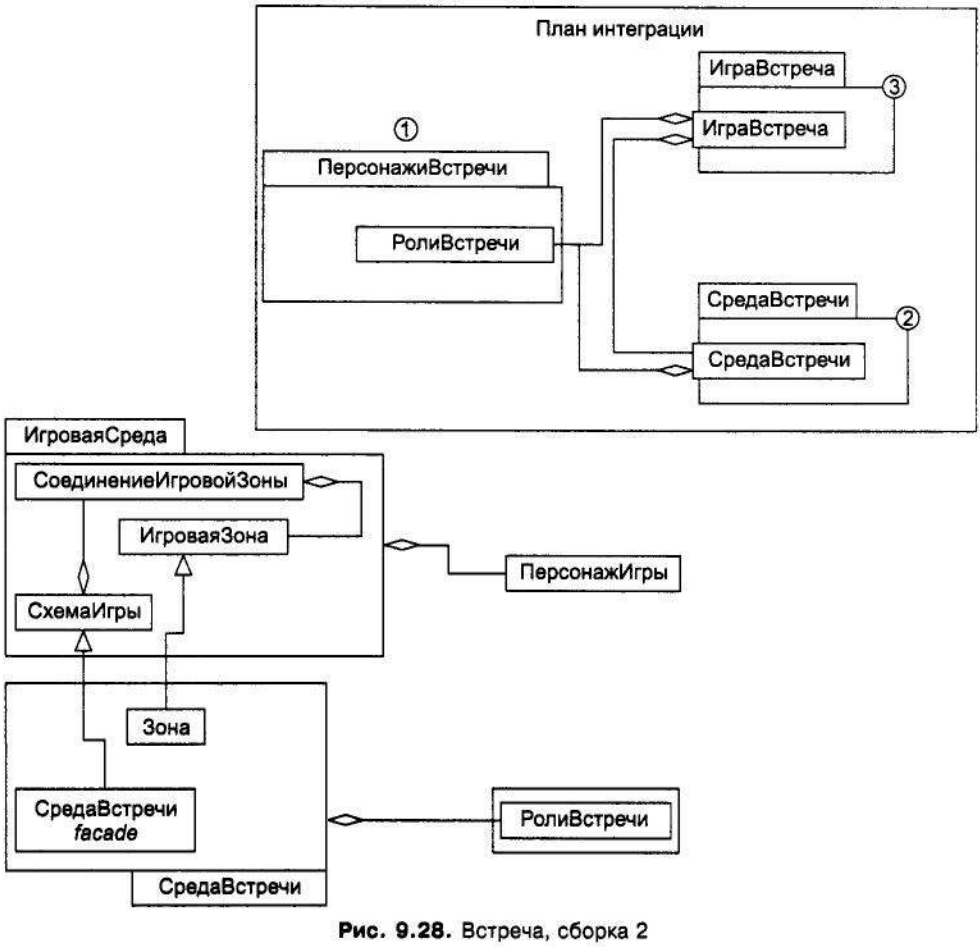


Рис. 9.28. Встреча, сборка 2

Рис. 9.29. Статус Встречи после второй сборки

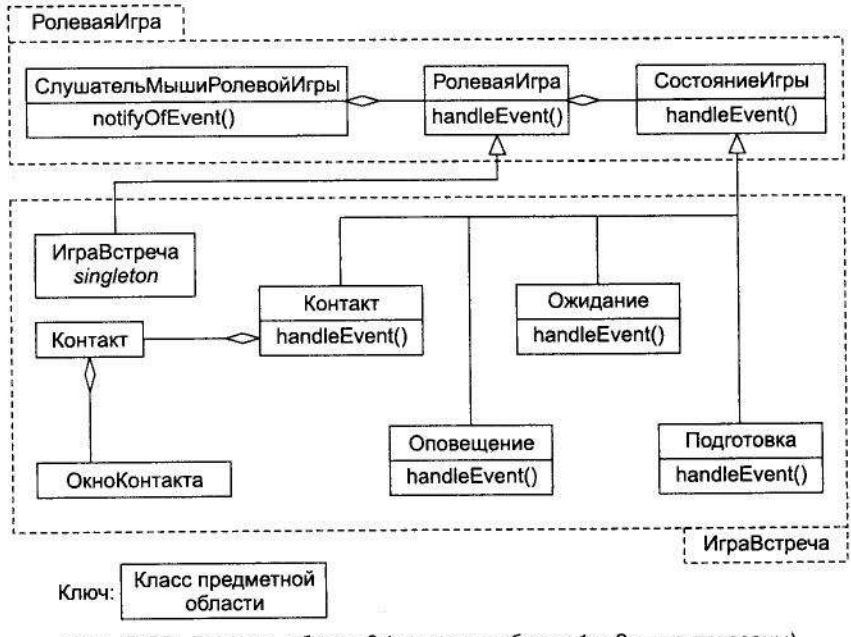


Рис. 9.30. Встреча, сборка 3 (содержит сборки 1 и 2 — не показаны)

Пример 2. Документация по тестированию программного обеспечения (STD) для видеоигры Встреча

История версий этого документа:

1 ноября 1998. Э. Брауде: начальный черновик.

4 апреля 1999. Э. Брауде: внесены исправления.

23 августа 1999. Р. Боствик: просмотрены документы, даны рекомендации.

24 августа 1999. Э. Брауде: рекомендации выполнены.

Статус: следует закончить.

[Примечание для студентов. Этот документ описывает общее тестирование видеоигры Встреча. Документ использует структуру IEEE STD (введение, план, проектирование, тестовые варианты, процедуры, отчет, журнал учета, отчет о происшествиях, итоги) и касается различных частных тестов (интегральные тесты, системные тесты, приемосдаточные тесты и т. д.). Они, в свою очередь, описываются с использованием тех же заголовков IEEE STD.]

1. Введение

Данный документ содержит STD для видеоигры *Встреча* и каркас ролевой игры. Категории тестирования, которые затрагивает этот документ, включают в себя

модульное, интегральное, системное, приемосдаточное и инсталляционное тестирование. Этот документ описывает тестирование, необходимое для валидации первых трех сборок видеоигры *Встреча*. На каждом уровне тестирования используется стандарт IEEE 829-1983 для STD.

Принцип тестирования для видеоигры *Встреча* приведен в табл. 9.3.

Таблица 9.3. Подходы и документация тестирования различных типов

Тип тестирования	Подход	Соответствующие разделы документов
Модульное	Методы «белого» и «черного ящика»; тестирование методов и классов; тестирование на соответствие D-требованиям и проекту	Разделы SRS: 3.2. Классы/Объекты. Разделы SDD: 6. Детальное проектирование
Интегральное	Метод «серого ящика»; преимущественно пакетный уровень; ориентирован на сборки 1, 2 и 3; тестирование на соответствие архитектуре и C-требованиям	Разделы SRS: 2. Общее описание; 3.1. Требования к внешнему интерфейсу. Валидировать представительные требования в «3.2. Классы/Объекты». Разделы SDD: 3. Описание декомпозиции; 4. Описание зависимостей; 5. Описание интерфейсов
Системное	Метод «черного ящика»; все пакеты; вся система (сборка 3); тестирование на соответствие нефункциональным требованиям, архитектуре и C-требованиям	Разделы SRS: 2. Общее описание; 3.1. Требования к внешнему интерфейсу. Валидировать представительные требования в «3.2. Классы/Объекты»; 3.3. Требования к производительности; 3.4. Ограничения проектирования; 3.5. Атрибуты программной системы; 3.6. Дополнительные требования Разделы SDD: 3. Описание декомпозиции; 4. Описание зависимостей; 5. Описание интерфейсов; Валидировать представительные требования в «6. Детальное проектирование»
Приемосдаточное	Метод «черного ящика»; все пакеты; вся система (сборка 3); тестирование на соответствие C-требованиям и D-требованиям	Разделы SRS: 2. Общее описание; 3.2. Классы/Объекты
Инсталляционное	Метод «черного ящика»; все пакеты; вся система (сборки для конкретных конфигураций заказчика); тестирование на соответствие C-требованиям и D-требованиям	Разделы SRS: 2. Общее описание; 3.2. Классы/Объекты

[Примечание для студентов. Это применение рис. 9.5 к видеоигре Встреча. SDD не считается документом с требованиями к проектированию, но предъявляет требования к реализации. Иногда эти требования формулируются в отдельном документе. В примере, рассматриваемом в этой книге, не имеется отдельного документа, содержащего эти требования.]

2. Документация тестирования видеоигры Встреча

STD для видеоигры *Встреча* и каркаса ролевой игры охватывает план тестирования, спецификацию и отчетность. Существуют отдельные планы тестирования для разных типов тестирования (модульного, интегрального, системного, приемосдаточного и инсталляционного). Каждый план тестирования ссылается на свои спецификации проекта тестирования, тестовых вариантов и тестовых процедур. Отчетная документация по тестированию состоит из журнала испытаний, отчета о происшествиях и итогового отчета.

2.1. STD модульного тестирования

См. отдельный документ по модульному тестированию.

[Примечание для студентов. См. пример в главе 8.]

2.2. STD интегрального тестирования

Документация интегрального тестирования состоит из отдельных документов для сборок 1, 2 и 3, как будет описано далее. См. приложение А к SCMP для создания базиса интеграции.

2.2.1. STD сборки 1

2.2.1.1. План тестирования сборки 1

2.2.1.1.1. Идентификатор плана тестирования: Сборка1_ПТ

2.2.1.1.2. Введение

Данный план тестирования охватывает интегральные тесты для каркасного пакета *ПерсонажиИгры* и пакета *ПерсонажиВстречи*. Он описывает, как проверить, что персонаж игрока и внешний персонаж можно вызвать, модифицировать и показать с помощью одиночного объекта *РолиВстречи*.

2.2.1.1.3. Тестовые элементы

Классы и методы из пакетов *ПерсонажиИгры* и *ПерсонажиВстречи* тестируются через объект *РолиВстречи*.

2.2.1.1.4. Свойства, подлежащие тестированию

Свойства, тестируемые согласно спецификации проекта тестирования *Сборка1_СП*, основываются на требованиях SRS и SDD (табл. 9.4).

Таблица 9.4. Свойства, подлежащие тестированию в сборке 1

Документ	Раздел	Название требования
SRS	2.1.2.2	Концепция пользовательского интерфейса для установки значений характеристик
	3.2.ПВ	Персонаж Встречи
	3.2.ВП	Внешние персонажи
	3.2.ПИ	Персонажи игрока
	3.2.ХИ	Окно характеристик игрока
SDD для каркаса ролевой игры	3.1.2	Пакет Персонажи
	5	Описание интерфейса
SDD для игры Встреча	3.1.2	Пакет ПерсонажиВстречи
	4.2	Межпроцессные зависимости
	5.1.2	Интерфейс пакета ПерсонажиВстречи

2.2.1.1.5. Свойства, не подлежащие тестированию

[Примечание для студентов. Существует бесконечно много вопросов, которые не тестируются, однако иногда определение некоторых конкретных вопросов, не подлежащих тестированию, помогает прояснить процесс тестирования.]

Тестирование характеристик, ассоциирующихся с пакетами *СредаВстречи* и *ИграВстреча* и их каркасами, откладывается до интегрального тестирования сборок 1 и 2.

2.2.1.1.6. Подход

Подход в верификации сборки 1 состоит из проверки того, что все персонажи игры можно вызвать и показать с помощью объекта *РолиВстречи*. Тесты методов и интерфейсов проверяют, доступны ли необходимые открытые (`public`) методы интерфейсов пакета *ПерсонажиВстречи* объекту *РолиВстречи*.

2.2.1.1.7. Критерий успешного прохождения теста

Критерий успешного прохождения основан на удовлетворении соответствующих требований в SRS и SDD.

2.2.1.1.8. Критерий приостановки и требования продолжения

(Отсутствуют.)

2.2.1.1.9. Тестовые отчеты, подлежащие сдаче

Документы, перечисленные в табл. 9.5, следует сдать группе управления конфигурациями по завершении интегрального тестирования сборки 1.

Таблица 9.5. Идентификаторы документов тестирования

Документ	Идентификатор документа
Сборка 1: План тестирования	Сборка1_ПТ
Сборка 1: Спецификация проекта тестирования	Сборка1_СП
Сборка 1: Спецификации тестовых вариантов	Сборка1_ТВ1—Сборка1_ТВn

продолжение

Таблица 9.5 (*продолжение*)

Документ	Идентификатор документа
Сборка 1: Спецификации тестовых процедур	Сборка1_ТП1—Сборка1_ТПn
Сборка 1: Журналы тестовых испытаний	Сборка1_ЖТ1—Сборка1_ЖТn
Сборка 1: Отчет о происшествиях во время тестирования	Сборка1_ОП1—Сборка1_ОПn
Сборка 1: Итоговый отчет	Сборка1_ИО1—Сборка1_ИОn

2.2.1.1.10. Задачи тестирования

Задачи состоят из следующих шагов.

1. Загрузить сборку 1 и пакет *Build_1.*
2. Выполнить тестовые процедуры сборки 1 из метода `main()` в *Build_1Test* в пакете *Build_1.*
3. Составить отчетную документацию по тестированию в соответствии с разделом 2.2.1.1.9.
4. Сохранить всю документацию по тестированию и данные в соответствии с разделом 2.2.1.1.9 в управлении конфигурациями.

2.2.1.1.11. Требования среды

В зависимости от доступности оборудования можно использовать рабочие станции IBM PC, Sun SPARC или аппаратную конфигурацию Apple IMAC. Для тестирования сборки 1 может использоваться интерактивная среда разработки IBM Visual Age.

2.2.1.1.12. Ответственность

Салли Сильвер и Джоз Хернандес из группы контроля качества (SQA) отвечают за управление, подготовку и проведение интегрального тестирования сборки 1. Кроме того, группе разработчиков видеоигры *Встреча* направляются технические вопросы и отчеты о происшествиях во время тестирования. Управление конфигурациями сохраняет всю документацию по тестированию и данные.

2.2.1.1.13. Требования к персоналу и обучение

SPMP определяет общие потребности в персонале и тренинге для интегрального тестирования.

2.2.1.1.14. План

План интегрального тестирования включен в раздел 5.5 версий 5 и выше SPMP. (В разделе 5.5.5 обсуждается обновление SPMP для поддержания его соответствия выбранной архитектуре.)

[Примечание для студентов. В примерах этой книги не приводится обновленный SPMP.]

2.2.1.1.15. Риск и непредвиденные обстоятельства

Если группа контроля качества не может выполнить тесты или ошибки приводят к неприемлемому количеству сбоев в системе, группе Альфреда Муррея из команды разработчиков видеоигры *Встреча* будет поручено провести интегральное тестирование сборки 1.

2.2.1.1.16. Утверждение

Для завершения этой части тестирования требуется подписание утверждающего документа руководителем группы контроля качества, менеджером по разработке видеоигры *Встреча* и представителем группы управления изменениями.

2.2.1.2. Проект тестирования сборки 1

2.2.1.2.1. Идентификатор спецификации проекта тестирования

Следует определить.

2.2.1.2.2. Свойства, подлежащие тестированию

Тест для сборки 1 получает объект *РолиВстречи*, *ПерсонажИгрока* и *ВнешнийПерсонаж*, изменяет значения различных характеристик, получает эти значения, проверяет их корректность.

2.2.1.2.3. Обновления в подходе

Следует определить.

2.2.1.2.4. Идентификация теста

Следует определить.

2.2.1.2.5. Критерий успешного прохождения тестирования свойств

Следует определить.

2.2.1.3. Тестовые варианты сборки 1

2.2.1.3.1. Идентификатор спецификации тестовых вариантов

Следует определить.

2.2.1.3.2. Тестовые элементы

Функциональность, которую следует протестировать, находится в спецификации следующих открытых методов объекта *РолиВстречи*:

```
EncounterCast getTheEncounterCast()
GameCharacter getThePlayerCharacter()
GameCharacter gethTheForeignCharacter()
void setPlayerChracterQuality(String quality, float value)
void setForeignCharacterQuality(String quality, float value)
float getPlayerCharacterQuality()
float getForeignCharacterQuality()
```

Их следует протестировать в соответствии с табл. 9.6.

Таблица 9.6. Входные-выходные данные и действия интегральных тестов

	Характеристика	Входное значение игрока	Входное значение внешнего персонажа	Другие	Действие
B1.1	Отсутствует	Отсутствует	Отсутствует	Получить персонаж игрока	Проверить по имени

продолжение

Таблица 9.6 (*продолжение*)

	Характеристика	Входное значение игрока	Входное значение внешнего персонажа	Другие	Действие
B1.2	Отсутствует	Отсутствует	Отсутствует	Получить внешний персонаж	Проверить по имени
B1.3	Сосредоточенность	30	40	Отсутствует	Проверить, что выходные данные равны входным
B1.4	Выносливость	30	40	Отсутствует	Проверить, что выходные данные равны входным
B1.5	...	...	...	...	...

2.2.1.3.3. Спецификация входных данных

См. табл. 9.6.

2.2.1.3.4. Спецификация выходных данных

См. табл. 9.6.

2.2.1.3.5. Требования к среде

Это тестирование выполняется только для пакетов *ПерсонажиИгры* и *ПерсонажиВстречи*.

2.2.1.3.6. Специальные требования к процедурам

Нет.

2.2.1.3.7. Зависимость от интерфейса

Нет.

[Примечание для студентов. Этот раздел описывает связь между разными интерфейсами. Это будет важно для будущих сборок, но не для первой сборки.]

2.2.1.4. Тестовые процедуры сборки 1

2.2.1.4.1. Идентификатор спецификации тестовых процедур

[Примечание для студентов. Указывает на класс (метод), из которого следует запускать тест.]

Integration_Tests/Build1_Test в пакете Tests.

2.2.1.4.2. Цель

Установить тест сборки 1 с минимальным количеством других частей программы.

2.2.1.4.3. Специальные требования

Следует создать тест в Integration_tests/Build1_Test, состоящий из класса с одним методом `main()`. Тесты 1, 2, 3... следует выполнить, а полученные результаты сравнить.

2.2.1.4.4. Процедурные шаги

Разместите входные данные и ожидаемые результаты характеристик в файле Build1_test_data в следующем формате:

<название характеристики> <входные данные> <ожидаемый результат>
<название характеристики> <входные данные> <ожидаемый результат>
...

Не должно быть никакого дополнительного текста в начале или в конце.

2.2.1.5. Отчет о проведении тестирования сборки 1

2.2.1.5.1. Идентификатор отчета о проведении тестирования

Сборка1_ИО1

2.2.1.5.2. Посланные элементы

Итоговый отчет о тестировании, журнал испытаний, отчет о происшествиях.

2.2.1.5.3. Местоположение

Документация для сборочного теста 1 находится...

2.2.1.5.4. Статус

Еще не выполнен.

2.2.1.5.5. Утверждения

Тестирование сборки 1 должно быть утверждено менеджером контроля качества.

2.2.1.6. Журнал испытаний сборки 1

2.2.1.6.1. Идентификатор журнала испытаний

Сборка1_ЖТ1

2.2.1.6.2. Описание

Этот документ описывает результаты тестирования сборки 1 (пример в табл. 9.7).

Таблица 9.7. Журнал испытаний сборки 1

Номер теста	Результат	Ссылка на ошибку
1	Пройден	Отсутствует
2	Не пройден	1823
3	Потеря данных — повторить	Отсутствует
4	Потеря точности в возвращенном значении	2872
5	...	...

2.2.1.6.3. Записи действий и событий

[Примечание для студентов. «Ссылка на дефект» — это число, используемое системой отслеживания дефектов для данного конкретного дефекта.]

2.2.1.7. Отчет о происшествиях во время тестирования сборки 1

2.2.1.7.1. Идентификатор отчета о происшествиях во время тестирования

Сборка1_ОПЗ

2.2.1.7.2. Итоги

См. табл. 9.7.

2.2.1.7.3. Описание происшествий

Эд Блэйк отвлекся при выполнении теста 3 на сработавшую в здании сигнализацию и не смог записать результаты теста. Было решено не прерывать и не повторять тестовую последовательность и включить тест 3 в тестирование для сборки 2.

2.2.1.7.4. Влияние

Было решено, что события, описанные выше, недостаточно серьезны для того, чтобы повторно запускать этот тест.

2.2.1.8. Итоговый отчет о тестировании сборки 1

2.2.1.8.1. Идентификатор итогового отчета о тестировании

Следует определить.

2.2.1.8.2. Итоги

Тестирование сборки 1 прошло успешно, за исключением отмеченных дефектов. Они будут обработаны в обычном процессе исправления дефектов.

2.2.1.8.3. Конфликты

См. отчет о происшествиях во время тестирования сборки 1.

2.2.1.8.4. Полная оценка

Будет приложено.

[Примечание для студентов. Дополнительные примечания, подробности.]

Нет.

2.2.1.8.5. Подведение итогов по результатам

Будет приложено.

2.2.1.8.6. Оценка

Будет приложено.

2.2.1.8.7. Подведение итогов действиям

Будет приложено.

2.2.1.8.8. Утверждения

_____ Утверждаю Не утверждаю

Менеджер контроля качества

2.2.2. STD сборки 2

(Формат аналогичен формату STD для сборки 1.)

2.2.2.1. План тестирования сборки 2

Эти тесты будут проверять, что все зоны игры можно вызвать и показать через объект *СредаВстречи* и что соединения между зонами согласуются с SRS.

2.2.2.2. Проект тестирования сборки 2

Эти тесты будут прежде всего проверять получение корректного объекта *СредаВстречи*, а затем показывать, что объекты *Зона* и *СоединениеЗоны* можно получить по требованию.

2.2.2.3. Тестовые варианты сборки 2

Тестируемая функциональность содержится в приведенных ниже открытых функциях класса *СредаВстречи*.

```
GameArea getTheDressingRoom()
GameArea getTheDungeon()
...
EncounterEnvironment getTheEncounterEnvironment()
```

2.2.2.4. Тестовые процедуры сборки 2

Следует добавить.

2.2.2.5. Поэлементный отчет о проведении тестирования сборки 2

Следует добавить.

2.2.2.6. Журнал испытаний сборки 2

Следует добавить.

2.2.2.7. Отчет о происшествиях во время тестирования сборки 2

Следует добавить.

2.2.2.8. Итоговый отчет о тестировании сборки 2

Следует добавить.

2.2.3. STD сборки 3

Следует добавить.

2.3. STD системного тестирования

[Примечание для студентов. Вспомните (см. рис. 9.6), что системные тесты проверяют корректность реализации архитектуры.]

2.3.1. План системного тестирования

[Примечание для студентов. Эти тесты проверяют архитектуру (рис. 9.31).]

Эти тесты проверяют, что результаты действий в игре *Встреча* корректно проявляют себя в виде перемещений персонажей в среде.

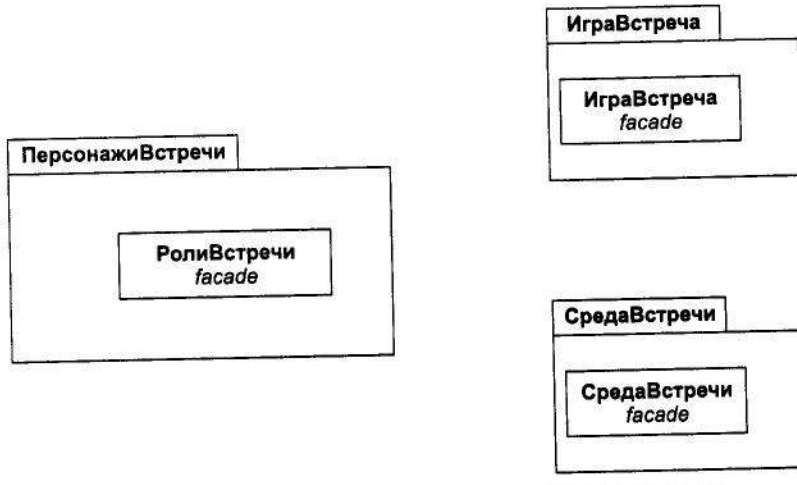


Рис. 9.31. Архитектура модуляризации видеоигры Встреча

2.3.2. Проект системного тестирования

[Примечание для студентов. Системные тесты разработаны для верификации архитектуры путем выполнения и проверки последовательностей интерфейсных методов.]

2.3.3. Тестовые варианты

Системный тест 1.

1. Переместить персонаж игрока в подвал.
2. Переместить внешний персонаж во двор.
3. Переместить внешний персонаж в подвал.
4. Выполнить встречу в подвале.

Системный тест 2.

1. ...

2.3.4. Тестовые процедуры

Системные тесты выполняются из пакета *SystemTest*. Системный тест *N* выполняется посредством метода `main()` класса *SystemTestN*.

2.4. STD приемосдаточного тестирования

2.4.1. План приемосдаточного тестирования

[Примечание для студентов. Интегральные тесты проверяют, что требования к игре Встреча удовлетворены согласноSRS.]

Приемосдаточные тесты хранятся в пакете *AcceptanceTest* и содержат варианты использования.

Вариант использования «Инициализировать» показан на рис. 9.32 и выполняется из метода `main()` класса *Initialize* пакета *AcceptanceTest*.

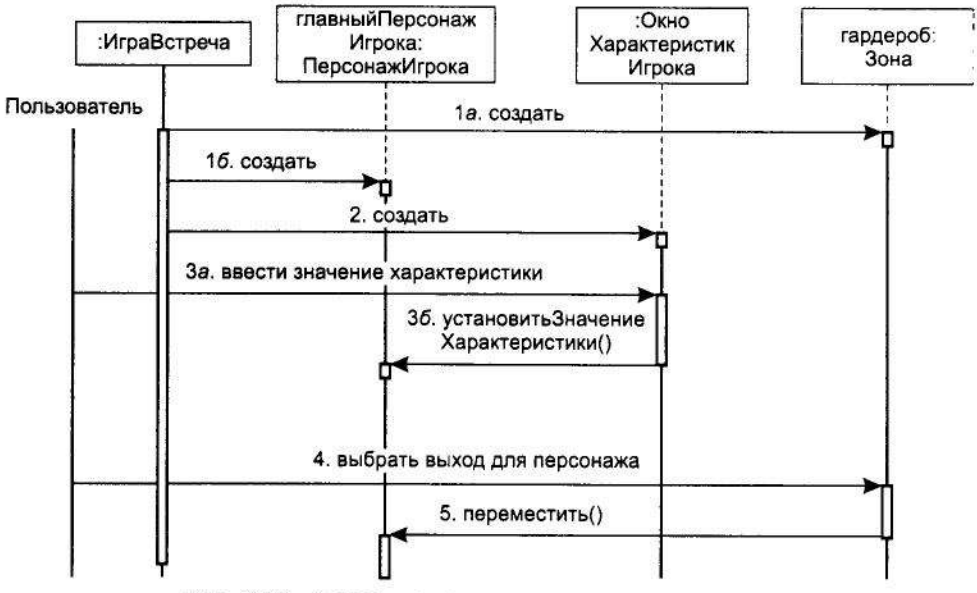


Рис. 9.32. Диаграмма последовательности для варианта использования «Инициализировать»

Вариант использования «Встретить внешний персонаж» показан на рис. 9.33 и выполняется из метода `main()` класса *AcceptanceTest.Initialize*.

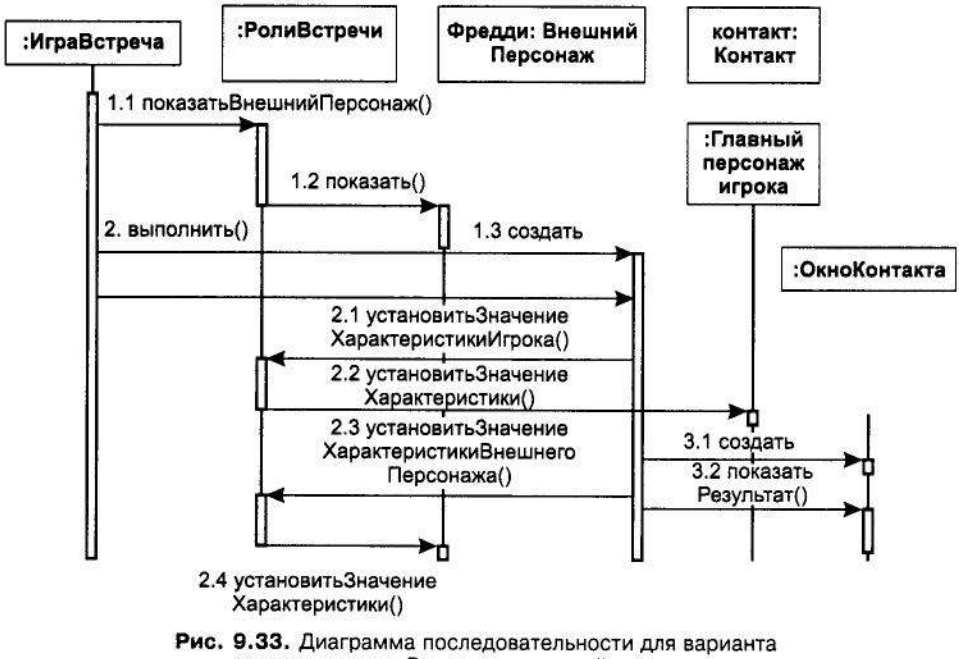


Рис. 9.33. Диаграмма последовательности для варианта использования «Встретить внешний персонаж»

2.4.2. Проект приемосдаточного тестирования

Варианты использования, упомянутые в разделе 2.4.1, должны быть выполнены последовательно несколько раз в соответствии с тестовыми вариантами из раздела 2.4.3.

2.4.3. Тестовые варианты

2.4.3.1. Тестовые варианты для варианта использования «Инициализировать»

[Примечание для студентов. Тесты являются экземплярами варианта использования «Инициализировать», называемыми также «сценариями».]

2.4.3.1.1. Приемосдаточный тест 1 «Инициализировать»

Запустить игру.

Назначить главному персонажу следующие значения характеристик в указанном порядке:

- Сила: 30
- Сосредоточенность: 20

Переместить главный персонаж во двор.

2.4.3.1.2. Приемосдаточный тест 2 «Инициализировать»

Запустить игру.

Назначить главному персонажу следующие значения характеристик в указанном порядке:

- Сила: 30
- Сосредоточенность: 20
- Терпение: 30

Переместить главный персонаж во двор.

2.4.3.1.3. Приемосдаточный тест 3 «Инициализировать»

Запустить игру.

Назначить главному персонажу следующие значения характеристик в указанном порядке:

- Сила: 30
- Сосредоточенность: 20

Переместить главный персонаж в подвал.

2.4.3.1.4. Приемосдаточный тест 4 «Инициализировать»

...

2.4.3.2. Тестовые варианты для варианта использования «Встретить внешний персонаж»

2.4.3.2.1. Приемосдаточный тест 1 «Встретить внешний персонаж»

Установить значение терпения главного персонажа равным 30.

Установить значение терпения внешнего персонажа равным 20.

Переместить главный персонаж в гостиную.
Заставить внешний персонаж войти в гостиную.
Убедиться, что контакт имел место.
Пронаблюдать окно контакта с результатами.
(Значение терпения игрока должно стать 40, внешнего персонажа — 10).

2.4.3.2.2. Приемосдаточный тест 2 «Встретить внешний персонаж»

Установить значение силы главного персонажа равным 30.
Установить значение силы внешнего персонажа равным 20.
Переместить главный персонаж в подвал.
Заставить внешний персонаж войти в подвал.
Убедиться, что контакт имел место.
Пронаблюдать окно контакта с результатами.
(Значение силы игрока должно стать 40, внешнего персонажа — 10).

2.4.3.2.3 Приемосдаточный тест 3 «Встретить внешний персонаж»

...

2.4.4. Тестовые процедуры

Приемосдаточные тесты будут проводиться двумя представителями заказчика. Эти представители будут проводить тестирование при участии представителя фирмы-разработчика. По возможности события, такие как появление внешнего персонажа в зоне, должны возникать как результат случайных процессов игры, а не эмуляции. Журнал испытаний должны вести представители заказчика. По окончании тестов он должен быть подписан всеми сторонами: любая сторона может добавить в журнал свои возражения.

2.5. STD тестирования инсталляции

[Примечание для студентов. Эти тесты проверяют, корректно ли работает программа на требуемой аппаратной конфигурации и в среде операционной системы.]

Тестирование инсталляции видеоигры *Встреча* состоит из выполнения системных тестов на следующих аппаратных конфигурациях.

1. IBM-совместимый ПК, на котором имеется не менее 32 Мбайт RAM и 100 Мбайт дискового пространства.
2. Sun SPARC модель mmm не менее чем с 32 Мбайт RAM и 100 Мбайт дискового пространства.
3. Apple IMAC модель 1234 или более поздняя не менее чем с 32 Мбайт RAM и 100 Мбайт дискового пространства.

Глава 10
Сопровождение

Положение темы настоящей главы в контексте разработки программного обеспечения иллюстрирует рис. 10.1.

- Разделы 10.1–10.7
- Упражнения
- Пример: сопровождение проекта *Встреча*

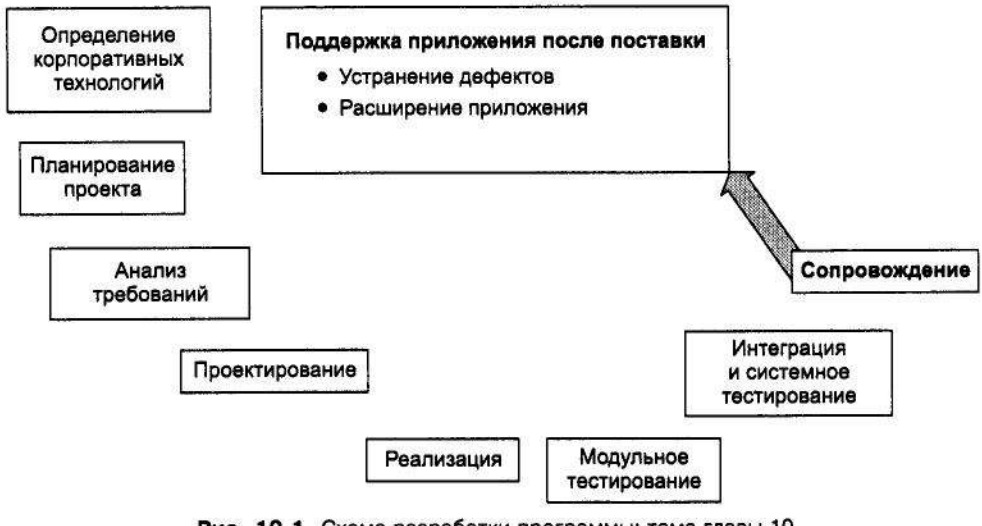
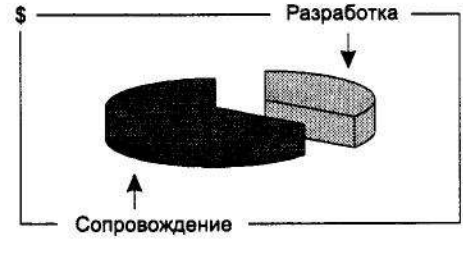


Рис. 10.1. Схема разработки программы: тема главы 10

Вопросы, которые рассматриваются в этой главе.

- Понимание определения *сопровождения программ.*
- Правильная оценка затрат на сопровождение.
- Понимание вопросов сопровождения программ.
- Организация сопровождения.



- Использование стандарта сопровождения IEEE.
- Применение метрик сопровождения.

10.1. Введение

Рассмотрим типичную последовательность действий по обработке *запроса на сопровождение* (MR — maintenance request).

1. Будьте готовы вести учет необходимых метрик, включая:
   - количество добавленных строк;
   - количество измененных строк;
   - время, ушедшее на подготовку, разработку, кодирование, тестирование.
2. Убедитесь в том, что запрос был подтвержден.
3. Тщательно изучите проблему:
   - воспроизведите ее или
   - уточните ситуацию иными средствами.
4. Классифицируйте запрос на сопровождение как исправление или усовершенствование.
5. Решите, требует ли реализация переработки на более высоком уровне:
   - в случае положительного решения рассмотрите возможность объединения данного запроса с другими.
6. Произведите запрошенную модификацию (то есть внесите необходимые изменения).
7. Планируйте переход с учетом текущего состояния.
8. Контролируйте влияние небольших изменений в масштабе всего приложения:
   - небольшие изменения могут быть очень существенны!
9. Реализуйте изменения.
10. Выполните модульное тестирование измененных компонентов.
11. Выполните регрессионное тестирование,
    - убедитесь, что изменения не ухудшили достигнутые характеристики.
12. Выполните системное тестирование с новыми функциями.
13. Обновите документацию, описывающую конфигурацию, требования, проектирование и тестирование.

Далее эти действия будут рассмотрены более подробно с соответствующими теоретическими обоснованиями.

10.1.1. Сопровождение программ

Сопровождение программного продукта включает в себя все действия, выполняемые с приложением после поставки продукта заказчику. В словаре IEEE [57]

сопровождение программы определяется как процесс изменения программной системы или компонента после поставки с целью исправления ошибок, повышения производительности или иных параметров, а также для адаптации к изменившимся условиям.

По различным оценкам сопровождение программы составляет от 40 до 90 % стоимости всего жизненного цикла приложения (например, [31], [89]). Можно возразить, что в указанное выше определение сопровождения включены усовершенствования, которые лучше было бы считать дополнительными разработками. В любом случае объем работ по сопровождению программ обычно достаточно велик. Наиболее значительные усилия в области сопровождения были затрачены на решение проблемы 2000 года (Y2K). Изменение приложений для работы с датами, относящимися к новому тысячелетию, потребовало огромного труда. Эти работы относятся именно к сопровождению программ, потому что их целью было сохранение функциональности существующих приложений.

Лехман [Le1, Le2] предложил «закон», заключающийся в том, что *любая программа, адаптацией которой к изменяющимся условиям никто не занимается, с течением времени неизбежно теряет свою ценность*. Другими словами, если приложение остается неизменным, польза от него постепенно убывает.

10.1.2. Вопросы сопровождения программ

Чтобы представить, какие проблемы возникают при сопровождении программ, вообразите приложение настолько большое, что ни один человек не может знать всех его функций и особенностей. Автор работал с защитными системами больших кораблей, которые могут быть отнесены именно к этому классу программ. Большую проблему в сопровождении таких программ создает «эффект ряби», возникающей при внесении изменений. Когда множество относительно безвредных изменений производятся одновременно в «удаленных уголках» больших приложений, эти изменения могут приводить к повреждению приложения как целого.

Беннетт [8] упорядочил проблемы, связанные с сопровождением программ, разделив их на три категории.

- Проблемы управления:
  - трудность выявления прибыли на инвестированный капитал.
- Проблемы обработки:
  - для обслуживания потока запросов на сопровождение требуется жесткая координация.
- Проблемы технического характера:
  - трудность учета всех результатов изменений;
  - высокая стоимость тестирования по сравнению с пользой от отдельных изменений: идеальным было бы сосредоточенное тестирование, но оно стоит дорого. Невозможно полностью исключить регрессионное тестирование.

Руководство обычно гораздо сильнее интересуется процессом поставки приложений. Кроме того, затраты на сопровождение программы трудно классифицировать как прибыль на инвестированный капитал.

Раскроем этот момент подробнее. Предположим, что Военно-морской флот уведомляет нас (фирму-подрядчик) о том, что алгоритм согласования трех независимых источников навигационных данных работает неправильно. Необходимо оценить стоимость внесения исправлений. Пример расчета приведен в табл. 10.1. Затраты на изменение программы при стоимости одного человеко-часа $50–100 (с учетом пособий и т. п.) будут составлять, таким образом $5600–28 000. Разброс значений очень велик.

Таблица 10.1. Оценка стоимости выполнения запроса на сопровождение

Операция	Длительность, человеко-дни	Операция	Длительность, человеко-дни
1. Уточнение проблемы и выявление функций, подлежащих изменению или добавлению	2–5	6. Компиляция и интеграция в основное тело программы	2–3
2. Разработка необходимых изменений	1–4	7. Тестирование функциональности измененных компонентов	2–4
3. Анализ влияния факторов	1–4	8. Регрессионное тестирование	2–4
4. Внесение изменений в исходный код	1–4	9. Выпуск новой версии и отчет о результатах	1
5. Изменение SRS, SDD, STP, сведений о конфигурации		Итого	14–35

Если бы мы имели дело с одним конкретным изменением, проблемы были бы еще не так велики. Обычно организации сталкиваются с непрерывным потоком запросов на сопровождение. Большое количество изменений позволяет снизить стоимость каждого из них, но поток запросов создает повышенные требования к системе обработки. Усилия программистов, тестеров и разработчиков документации нужно координировать. Например, нужно ли обновлять SRS после обнаружения ошибки, после ее устранения, после тщательного тестирования или же, наконец, после объединения данного исправления с другими операциями, выполненными в рамках работ по сопровождению? В любом случае согласованность документации и исходного кода будет время от времени нарушаться. Без внимательного руководства эти кратковременные, казалось бы, несоответствия начнут размножаться, в результате чего документация перестанет соответствовать реальным функциям приложения, и никто не будет знать, что же на самом деле творится с приложением.

Суть в том, чтобы сделать изменения понятными и даже элегантными. Элегантность хорошо иллюстрирует следующий пример: можно увеличить полезное пространство в доме, пристроив к нему сарай, а можно сохранить цельность зда-

ния, увеличив его объем искусными архитектурными приемами. Пристройка может ухудшить внешний вид и структуру всего здания, а хороший архитектор всегда постарается этого избежать.

Тестирование — еще одна важная техническая проблема. Даже сосредоточенное тестирование измененных или добавленных компонентов отнимает достаточно времени, потому что для этого часто приходится разрабатывать специальные планы. Но сосредоточенного тестирования недостаточно, потому что возможность распространения «ряби», вызванной изменениями, требует проведения регрессионного тестирования, которое могло бы исключить ухудшение имевшейся функциональности приложения. Затраты на тестирование составляют значительную долю общих затрат на сопровождение программы.

10.1.3. Организация процесса сопровождения

Блок-схема организации процесса сопровождения согласно [89] приведена на рис. 10.2. Пункты 1*а*–1*д* фактически относятся к этапу разработки. «Разработка с учетом сопровождения» подразумевает попытку предугадать, в каких направлениях будут развиваться требования к продукту, и учесть это в плане. С этой целью часто применяются, например, образцы проектирования. «Удобство поддержки» в пункте 1*б* означает возможность эффективного сопровождения. Для этого в код должны включаться подробные комментарии. Персонал, занимающийся сопровождением, имеет обычно не столь специализированные знания, как разработчики, потому что сопровождение требует работы с гораздо большим объемом кода. Тем не менее служба сопровождения должна быть способна после внимательного изучения кода прийти к пониманию смысла каждой инструкции. Только после этого допустимо внесение изменений. Удобство поддержки гарантируется постоянным уровнем качества кода.

Пункт 2 состоит в принятии решения о масштабах затрат на сопровождение: главным образом речь идет о том, следует ли относить усовершенствования к работам по сопровождению. Возможные варианты рассматриваются в разделе 10.2.

Согласно пункту 3, вам предстоит выбрать, своими или наемными силами выполнять работы по сопровождению. Заключение контракта на сопровождение программы обладает преимуществами конкурентоспособной цены и дает возможность владельцу приложения сосредоточиться на других задачах. Недостаток в том, что ваши сотрудники постепенно теряют контроль над кодом приложения.

Планирование процесса сопровождения обсуждается в разделе 10.5.

10.2. Виды работ по сопровождению

Следует различать работы по сопровождению, направленные на *устранение дефектов* (fixing) и на *усовершенствование* (enhancing) приложения. Различные исследования показали, что 60–80 % работ обычно относится к усовершенствованию приложения, а не к исправлению его недостатков (например, [89] и [38]).

Лиенц, Свенсон и др. [79] дополняют иерархию работ, разбивая каждую из определенных выше категорий на две (рис. 10.3).

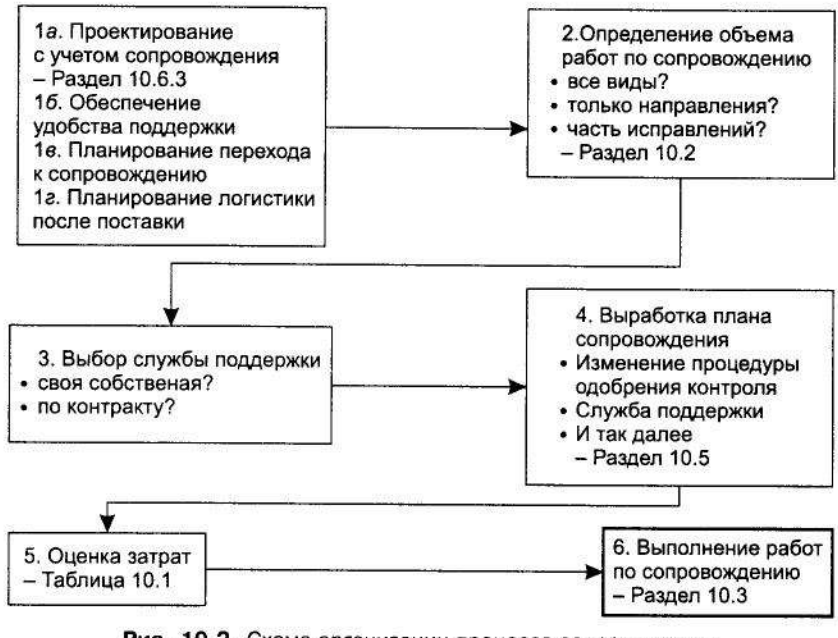


Рис. 10.2. Схема организации процесса сопровождения

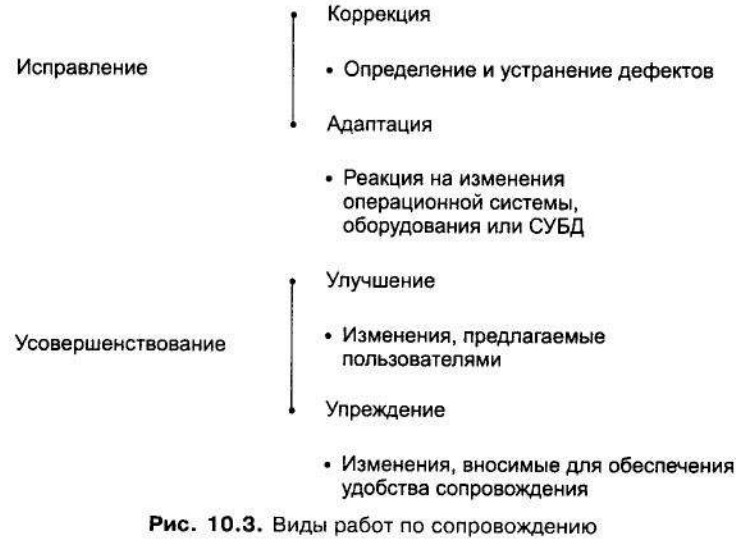


Рис. 10.3. Виды работ по сопровождению

Приспособление, или *адаптация* (adaptive maintenance), относится к исправлению недостатков, потому что функциональность приложения в результате не изменяется и никакого усовершенствования не происходит.

Необходимость *упреждающего* сопровождения (preventive maintenance) следует из наблюдений Лемаша [Le1, Le2]): *без специальных поправок структура сопровождаемой программы постепенно усложняется и со временем становится настолько сложной, что стоимость ее изменения становится неприемлемой.*

Рассмотрим пример одного из видов сопровождения применительно к игре *Встреча*. Пример исправления недостатков показан в запросе на сопровождение 78. Персоналу, который занимается сопровождением, нужно решить: кроется ли ошибка в методе `adjustQuality()` или неправильно сформулированы требования к персонажам игры. На самом деле проблема именно в требованиях, которые гласят:

- пользователь может установить для любой характеристики своего персонажа любое значение, не превышающее текущей суммы значений всех характеристик;
- пропорции всех остальных значений при этом сохраняются;
- значение характеристики не может лежать в интервале от 0 до 0,5;
- сумма значений характеристик остается постоянной.

Как следует из запроса 78, эти требования противоречивы и не могут быть удовлетворены одновременно. Поскольку проблема обнаружена в требованиях к программе, заказчик должен санкционировать внесение изменений. Можно, например, ослабить последнее требование (требование инвариантности суммы), записав его в форме

$0 \le$ (*сумма характеристик*)$' \le$ abs(*сумма характеристик* $- N$),

где N — количество характеристик, abs — абсолютное значение (модуль числа), а x' — значение x после изменения одной из характеристик игроком. После такого изменения правил игроку все равно будет невыгодно менять значения характеристик слишком часто и слишком сильно, потому что при этом могут теряться очки.

Запрос на сопровождение 78

При изменении игроком значения одной из характеристик сумма всех значений должна сохраняться, но этого не происходит. Например, если характеристики имеют следующие значения: сила = 10, терпение = 0,8, выносливость = 0,8 (сумма = 11,6), и игрок устанавливает силу равной 11, в результате его характеристики получают значения: сила = 11, терпение = 0 и выносливость = 0.

Теперь займемся запросом на сопровождение с *улучшением* (perfective maintenance). Предположим, что отдел маркетинга решил сделать игру более привлекательной, основываясь на том, что пользователи должны получать максимально ощутимую награду за свой уровень. Предлагается, чтобы при переходе игрока на следующий уровень облик игры в целом становился более качественным. Отдел оформления предоставит необходимые рисунки, а отдел сопровождения должен внести изменения, отражающие новые требования. Запрос 162 будет обсуждаться далее в тексте главы.

Запрос на сопровождение 162

Измените игру Встреча так, чтобы она начиналась с отображения зон и их соединений в согласованном стиле.

Когда игрок достигает второго уровня, все зоны и их соединения отображаются в усовершенствованном согласованном стиле, который доступен только для уровня 2, и т. п. Отдел оформления обеспечит необходимые изображения.

10.3. Методы сопровождения

10.3.1. Анализ влияния факторов

Последовательность обработки запросов на сопровождение состоит из анализа, проектирования и реализации, точно так же, как и обычная разработка. Существенным отличием является необходимость анализа влияния изменений на артефакты. Согласно исследованию Вейсса [110], 19 % дефектов в приложениях образуются на этапе определения требований, 52 % — на этапе проектирования и 7 % — в процессе программирования. Многие другие авторы утверждают, что доля дефектов, вызванных неправильной формулировкой требований, должна быть значительно выше. Влияние устранения дефекта на артефакты иллюстрирует рис. 10.4.

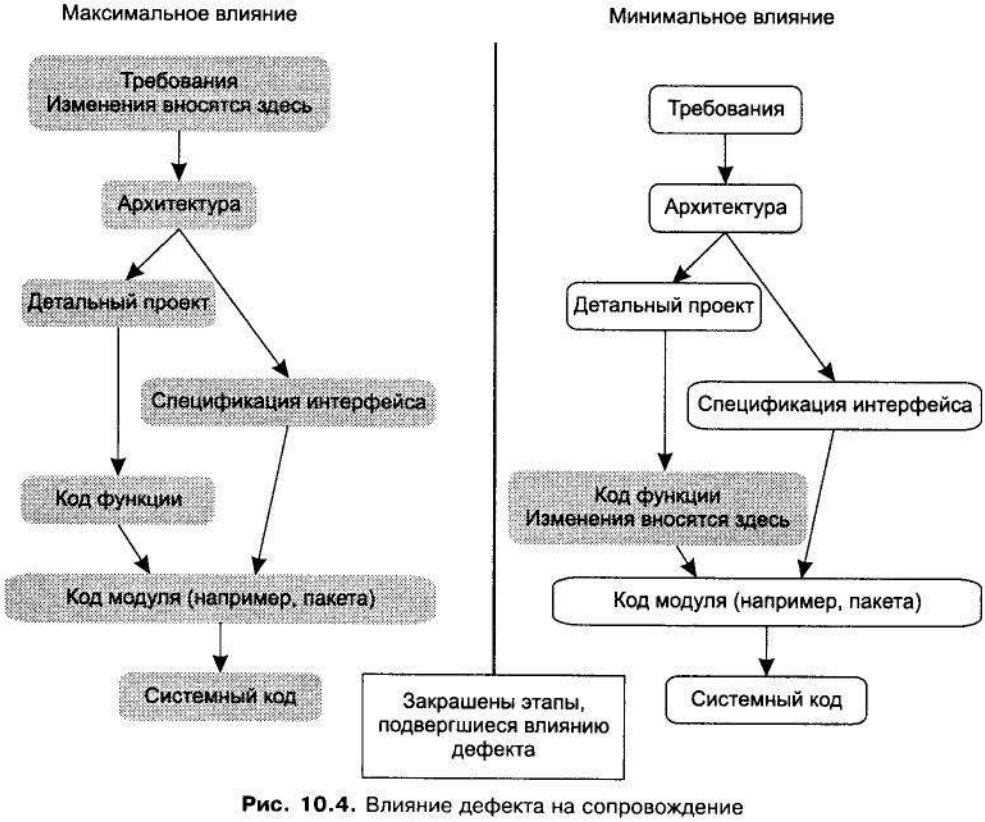


Рис. 10.4. Влияние дефекта на сопровождение

В случае минимального влияния изменения вносятся в один-единственный артефакт. Это происходит, например, при нарушении программистом стандарта именования локальных переменных или при удалении неиспользованной переменной из программы. Напротив, в худшем случае изменения распространяются на все этапы процесса. Даже для дефекта, возникшего на уровне кода (то есть дефекта, связанного лишь с неправильным кодированием), степень влияния может быть как малой, так и весьма значительной. Кажущееся простым изменение, например увеличение размера статического массива в C++, может вызвать сильную «рябь» по всему приложению.

Запрос 162, приведенный ранее, влияет на все аспекты процесса, показанного на рис. 10.5. Если приложение разрабатывается так, чтобы его можно было прослеживать, то документировать действия по сопровождению может быть относительно несложно. В противном случае последствия могут быть крайне дорогостоящими. Подробнее об анализе влияния действий по сопровождению на уровне кода рассказывается в книге [111].

Рис. 10.5. Влияние запроса 162

10.3.2. Обратное проектирование

История сопровождаемого программного продукта может быть очень непростой. Многие существующие приложения часто оказываются снабжены скудной или противоречивой документацией. Первым шагом в организации рациональных

работ по сопровождению таких приложений должно быть создание подробной согласованной документации. Для этого часто требуется восстановление архитектуры по исходному коду — *обратное проектирование* (reverse engineering). Существует несколько программных средств, помогающих выполнить эту задачу. Например, пакеты Rational Rose (Rational Corporation) и Together (Object International) позволяют создавать модели объектов из исходного кода на C++ и Java. Получающиеся в результате диаграммы можно использовать для анализа хода мыслей разработчика. Из-за неоднозначности трактовки диаграмм на UML и механичности процесса обращения обратное проектирование не позволяет полностью понять намерения разработчика. Когда разработчик рисует прямоугольники и связи UML, их геометрическое размещение (например, группировка) несет смысловую нагрузку, воспринимаемую только людьми.

Обратное проектирование является более общим средством, чем преобразование кода в проект, поскольку представляет собой процесс восстановления содержания какого-то этапа по артефактам последующего этапа. Получение намерений из структуры возможно не всегда. Отсылаем читателя к недокументированному коду в разделе 1.5.1. Из-за отсутствия комментариев сделать выводы о назначении кода просто невозможно.

10.3.3. Реинжиниринг

В 1980 году в книге Хаммера и Чампи [40] о реинжиниринге бизнес-процесса (BPR — Business Process Reengineering) компаниям предлагалось заново взглянуть на процесс производства ценностей для потребителей, после чего спроектировать его заново. Хаммер и Чампи основное внимание уделяли процессу как целому, а не отдельным его частям. Процесс начинается с входных данных, например заказов, заканчивается окончательной отгрузкой товаров или предоставлением услуг и включает все способствующие достижению цели элементы, такие как поддержка программ и вклад сотрудников. Реинжиниринг заставляет взглянуть на требования к программным приложениям как на часть требований к предприятию (компании, организации и т. п.) в целом. Получающиеся приложения иногда называются *корпоративными приложениями* (enterprise application), хотя обсуждаемая концепция используется и вне контекста реинжиниринга бизнес-процесса. Реинжиниринг обычно заставляет ответить на вопрос о том, как *должны* работать существующие приложения, а не о том, как они работают. Фактически, при реинжиниринге к бизнес-процессу применяются концепции системной разработки, обсуждавшиеся в главе 3.

Реинжиниринг относят к сопровождению, потому что он требует перепланирования приложений. Сравнение экспансивного развития с реинжинирингом на примере моста приводится на рис. 10.6. В первом случае мост укрепляется, а во втором — перестраивается для того, чтобы отвечать возросшим требованиям.

В качестве примера рассмотрим компанию, приступающую к реинжинирингу процесса обучения менеджеров. Реинжиниринг подразумевает изучение процесса

от начала до конца. Обычно оказывается непрактичным переписывать программы с нуля, поэтому берутся имеющиеся программы, которые затем перепроектируются так, чтобы удовлетворять изменившимся требованиям. Высокоуровневая схема интеграции видеоигры *Встреча* в систему обучения менеджеров показана на рис. 10.7. Результатом реинжиниринга станет адаптация игры *Встреча* для получения приложения, моделирующего менеджмент и оценивающего результаты.

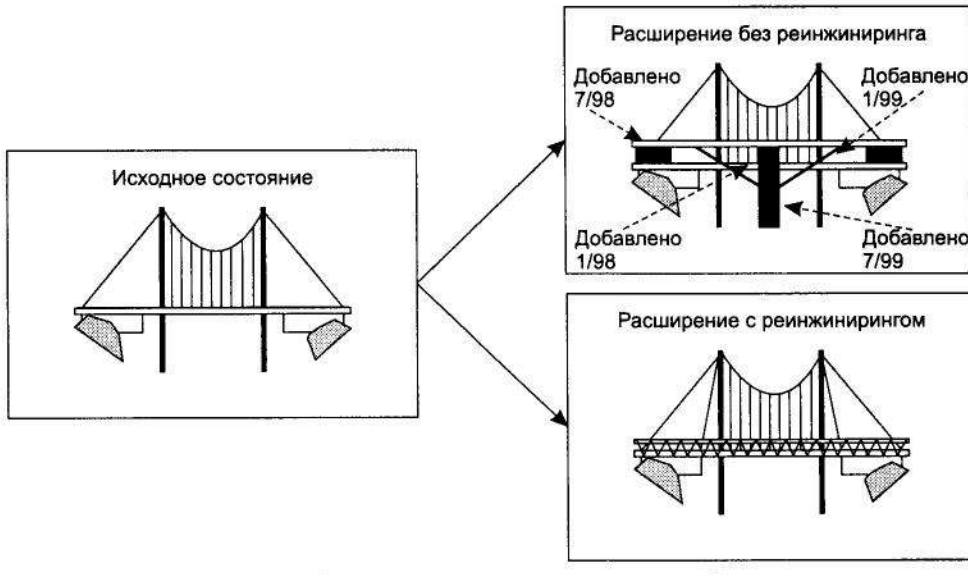


Рис. 10.6. Сопровождение с реинжинирингом и без него

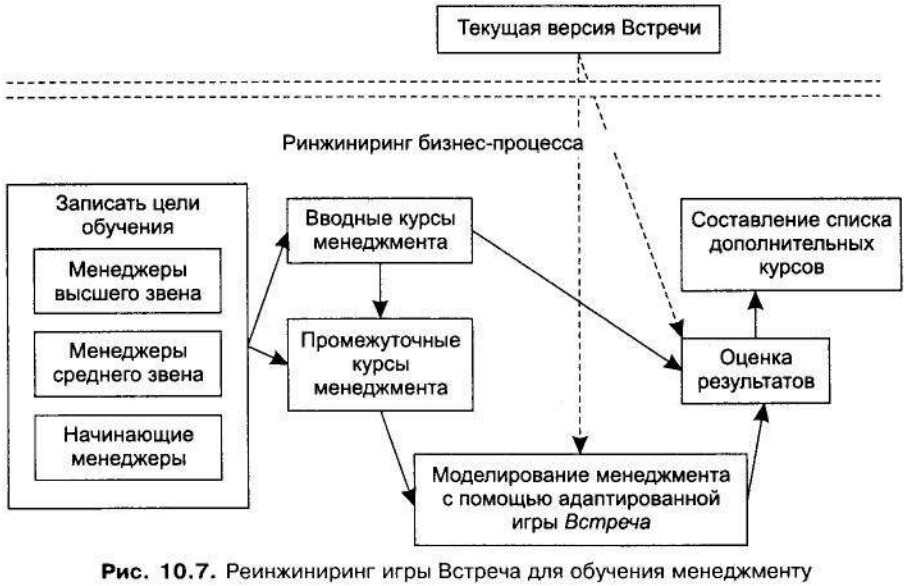


Рис. 10.7. Реинжиниринг игры Встреча для обучения менеджменту

10.3.3.1. Рефакторинг

Часто усовершенствование программы требует не просто изменения отдельных строк кода, а приложения гораздо больших усилий, не достигающих, однако, масштабов полного реинжиниринга. Иногда этот процесс называют *рефакторингом* (refactoring). Фаулер [32] рассматривает пример проекта для магазина видеофильмов — этот проект не слишком хорош, но вполне пригоден для конкретной простой задачи. Затем он показывает, как можно переделать проект под новые требования, например добавить возможность формирования новых типов отчетов. Этот процесс включает экстракцию метода (то есть создание метода, заменяющего имеющийся фрагмент кода). Другие примеры рефакторинга включают воплощение идей главы 7, посвященной реализации, в код, например замену литерной константы

```
final int MAX_NUM_VIDEOS = 18;
```

методом

```
static final int getMaxNumVideos() . . .
```

10.3.4. Унаследованные приложения

Унаследованные системы (legacy systems) — это приложения, решающие существующие задачи. Иногда термин *legacy* трактуется как *устаревшие* и применяется к программам, которые не стоят того, чтобы их модифицировать.

Беннетт [8] перечисляет возможные действия с унаследованными системами.

- Продолжать сопровождение;
- Прекратить сопровождение и:
  - заменить на покупной продукт;
  - заменить на собственный продукт, полученный обратным проектированием или разработкой с нуля. Возможна поэтапная замена;
  - присоединить к новому приложению. Сопровождение заморозить;
  - инкапсулировать и использовать как сервер. Возможно использование образца проектирования Adapter.

Присоединение и инкапсуляцию иллюстрирует рис. 10.8. Под меткой *i* на этом рисунке показано, каким образом новое приложение получается из исходного путем расширения или модифицирования последнего. В примере с игрой *Встреча* это могло бы потребоваться, если бы нам нужно было получить игру в реальном времени от первого лица. В такой ситуации к существующей игре *Встреча* можно было бы добавить графический процессор (расширение), который постоянно отображал бы все, что видит игрок, а исходный код изменить так, чтобы поле игры отображалось глазами игрока, а не в перспективе с видом сверху.

При инкапсуляции исходное приложение практически не изменяется. Новое приложение создается полностью независимо, а в процессе его выполнения вызывается функциональность унаследованного приложения. Это может осуществляться как напрямую (метка *ed*), так и через *обертку* (метка *ew*). Часть рисунка

под меткой *ed* отвечает образцу проектирования Adapter (см. главу 6), если унаследованное приложение является объектно-ориентированным. Обертка — это программное обеспечение, предоставляющее интерфейс для обращения клиентов к унаследованному приложению. В частности, обертка может сделать любое приложение внешне объектно-ориентированным. После этого можно применять образец проектирования Adapter.

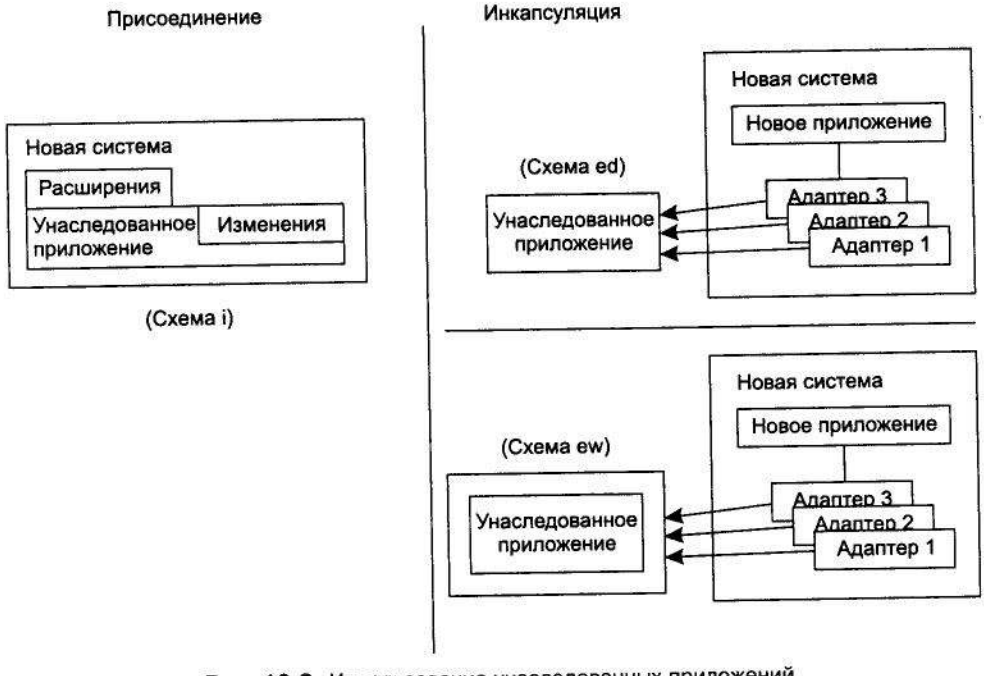


Рис. 10.8. Использование унаследованных приложений

10.3.5. Обновление документации

Действия по сопровождению включают в себя гораздо больше, нежели просто технические изменения и дополнения. Для отражения каждого такого действия требуется обновление всей цепочки документации. Например, исправление, вызванное дефектом в требованиях, приводит к изменению документации, содержащей требования к продукту, а также, вероятно, проектной документации и обязательно — документации на реализацию и тестирование. Кроме того, необходимо изменение состояния системы управления конфигурациями для отражения новой версии продукта. Для больших программ, в работе над которыми случалось принимать участие автору, устранение дефектов часто занимало меньше времени, чем обновление документации. Но если игнорировать необходимость обновления, то документация потеряет согласованность, из-за чего стоимость сопровождения начнет возрастать и в конце концов окажется просто неприемлемой.

10.4. Стандарт IEEE 1219-1992

Стандарт IEEE 1219-1992 определяет процесс сопровождения программного обеспечения. Семь стадий процесса, описанные в этом стандарте, приблизительно соответствуют стадиям процесса разработки. Каждая стадия характеризуется шестью атрибутами (рис. 10.9). Значения этих атрибутов для каждой из семи стадий процесса сопровождения приведены в табл. 10.2 и табл. 10.3. Содержание стандарта IEEE 1219-1992 приводится ниже.

1. **Определение задачи**
   - 1.1. Входные данные
   - 1.2. Процесс
   - 1.3. Контроль
   - 1.4. Выходные данные
   - 1.5. Факторы качества
   - 1.6. Метрики
2. **Анализ**
   - 2.1. Входные данные
   - 2.2. Процесс
     - 2.2.1. Анализ осуществимости
     - 2.2.2. Подробный анализ
   - 2.3–2.6. Контроль, Выходные данные, Факторы качества, Метрики
3. **Проектирование**
   - 3.1–3.6. Входные данные, Процесс, Контроль, Выходные данные, Факторы качества, Метрики
4. **Реализация**
   - 4.1. Входные данные
   - 4.2. Процесс
     - 4.2.1. Кодирование и тестирование
     - 4.2.3. Анализ и обзор рисков
     - 4.2.4. Проверка готовности к тестированию
   - 4.3–4.6. Контроль, Выходные данные, Факторы качества, Метрики
5. **Системное тестирование**
   - 5.1.–5.6. Входные данные, Процесс, Контроль, Выходные данные, Факторы качества, Метрики
6. **Приемосдаточное тестирование**
   - 6.1.–6.1. Входные данные, Процесс, Контроль, Выходные данные, Факторы качества, Метрики
7. **Поставка**
   - 7.1.–7.6. Входные данные, Процесс, Контроль, Выходные данные, Факторы качества, Метрики

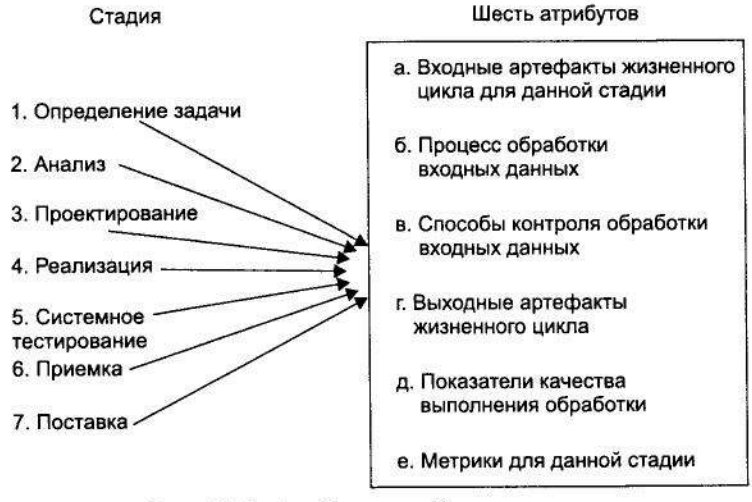


Рис. 10.9. Атрибуты стадий сопровождения

10.4.1. Определение задачи сопровождения

Стадия процесса обработки запросов на сопровождение, на которой происходит определение задачи, описана в табл. 10.2. В примере в конце этой главы определение задачи выполняется отделом маркетинга, который запросил и изучил жалобы пользователей на сложность алгоритма обмена характеристиками персонажей игры *Встреча*.

Таблица 10.2. Определение задачи запроса на сопровождение

	IEEE 1219-1992 **Стадия сопровождения 1: определение задачи**
а. Входные данные	Запрос на сопровождение
б. Процесс	Присвоить изменению номер
	Охарактеризовать по типу и степени серьезности
	Принять или отклонить изменение
	Выполнить предварительную оценку затрат
	Установить приоритет
в. Контроль	Присвоить запросу уникальный идентификатор
	Ввести запрос в хранилище
г. Выходные данные	Утвержденный запрос
д. Выбранные факторы качества	Ясность запроса
	Корректность запроса (например, тип)
е. Выбранные метрики	Количество упущений в запросе
	Количество поданных запросов к определенной дате
	Количество дублирующихся запросов
	Оценка времени на подтверждение проблемы

10.4.2. Анализ задачи

Стадия анализа задачи в процессе обработки запросов на сопровождение описана в табл. 10.3.

Таблица 10.3. Анализ запроса на сопровождение

	IEEE 1219-1992 **Стадия сопровождения 2: анализ задачи**
а. Входные данные	Исходная проектная документация
	Утвержденный запрос со стадии определения
б. Процесс	Изучить выполнимость запроса
	Исследовать влияние выполнения запроса
	Выполнить подробный анализ требуемых работ
	Уточнить содержание запроса
в. Контроль	Провести техническую проверку
	Проверить соответствие стратегии тестирования
	Проверить обновление документации
	Выявить вопросы, связанные с безопасностью и защитой
г. Выходные данные	Отчет о выполнимости
	Подробный отчет об анализе, в том числе о влиянии изменений
	Обновленные требования
	Предварительный список изменений
	План реализации
	Стратегия тестирования
д. Выбранные факторы качества	Понятность анализа
е. Выбранные метрики	Количество требований, подлежащих изменению
	Трудозатраты (на анализ запроса)
	Фактическая продолжительность

10.4.2.1. Пример анализа задачи сопровождения

Задачи сопровождения бывают как очень простыми, так и требующими предельного напряжения. Предположим, например, что мы хотим дополнить игру *Встреча* новыми параметрами изображения для главного игрока. Задача кажется довольно прямой и ясной, однако объем работ по запросам, подобным этому, часто недооценивается. Процедура анализа задачи должна выявлять реальные трудозатраты на внесение изменений и дополнений. Например, может выясниться, что увеличение параметров изображений может потребовать полного изменения методов выбора и отображения.

Проанализируем приведенный выше запрос на сопровождение 162 и оценим ресурсы на проектирование и реализацию необходимых изменений. Будем использовать образец проектирования Abstract Factory, описанный в главе 6. Для согласования и добавления новых классов требуется изменение объектной модели.

После этого нужно будет заменить все ссылки на Area и AreaConnection в клиентском коде (использующем новую конфигурацию), такие как

```
new Area()
new Connection()
```

вызовами типа

```
LevelNBuilder.getArea()
LevelNBuilder.getConnection()
```

Охарактеризуем эти изменения в метриках IEEE.

- Количество изменений в требованиях для запроса 162: между 140 и 450. Поскольку мы упорядочили требования по классам, эта величина включает в себя:
  - количество новых классов, подлежащих описанию: 60–90 (пусть в игре будет 20–30 уровней; для каждого из них образец проектирования Abstract Factory требует создания класса Factory и подклассов Area и AreaConnection);
  - количество новых методов: (2–5 на класс) × (60–90 классов) = 120–450.
- Оценочная величина трудозатрат по запросу 162: от 2,4 до 9 человеко-месяцев. Вычисляется через количество человеко-часов на каждое требование исходя из имеющихся данных по конкретному проекту. Например, если в исходном проекте было 300 требований и он был завершен за 6 человеко-месяцев, мы можем считать, что каждое требование требует 0,02 человеко-месяца. Отсюда для запроса 162 получается результат (120–450) × 0,02 = 2,4–9 человеко-месяцев. Природа классов позволяет уменьшить эту оценку, вероятнее всего до 3 человеко-месяцев.
- Оценка фактической продолжительности по запросу 162. Вычисляется по имеющимся фактическим данным точно так же, как и трудозатраты.

Для оценки правильнее использовать метод линейной регрессии, заключающийся в построении аппроксимирующей прямой по имеющимся фактическим данным.

10.4.3. Проектирование запроса на сопровождение

Стадия проектирования запроса на сопровождение описана в табл. 10.4. Например, разработка запроса 162 выполняется с применением образца проектирования Abstract Factory и состоит в модификации исходного пакета EncounnterEnvironment (СредаВстречи). Документация для этого пакета до его модификации показана на рис. 10.10.

Таблица 10.4. Проектирование запроса на сопровождение

	IEEE 1219-1992
	Стадия сопровождения 3: проектирование
а. Входные данные	Исходная проектная документация
	Анализ, полученный на предыдущей стадии

	IEEE 1219-1992 Стадия сопровождения 3: проектирование
б. Процесс	Создать тестовые варианты
	Просмотреть: требования и план реализации
в. Контроль	Верифицировать проект
	Проинспектировать: проект и тестовые варианты
г. Выходные данные	Измененные: список модификаций, детальный анализ и план реализации
	Обновленные: каркас проекта и планы тестирования
д. Выбранные факторы качества	Гибкость (проектирования)
	Прослеживаемость
	Возможность повторного использования
	Понятность
е. Выбранные метрики	Трудозатраты в человеко-часах
	Фактическая продолжительность
	Количество изменяемых приложений

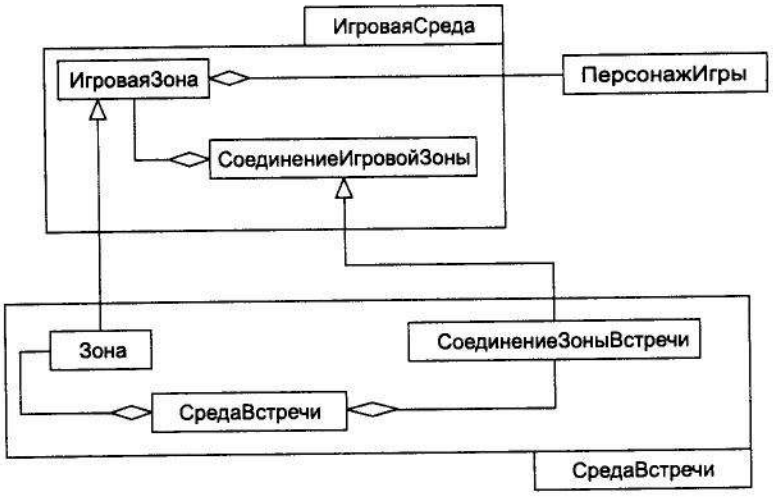


Рис. 10.10. Пакет СредаВстречи до модификации

В измененном приложении объекты `Area` и `EncounterAreaConnection` будут создаваться не непосредственно, а через методы `getArea()` и `getAreaConnection()`. Это методы нового класса `EnvironmentFactory`. Клиентская часть кода пакета `EncounnterEnvironment` может ничего не знать о типе создаваемых объектов `Area` и `AreaConnection`, поскольку все запросы на создание направляются через конкретный объект `EnvironmentFactory`, агрегируемый пакетом `EncounterEnvironment`. Во время выполнения программы клиентский код выбирает объект соответствующего подкласса `EnvironmentFactory`. На рис. 10.11 мы приводим классы `Area` и `AreaConnection` только для трех уровней игры, а не для всех уровней, запланированное число которых достаточно велико.

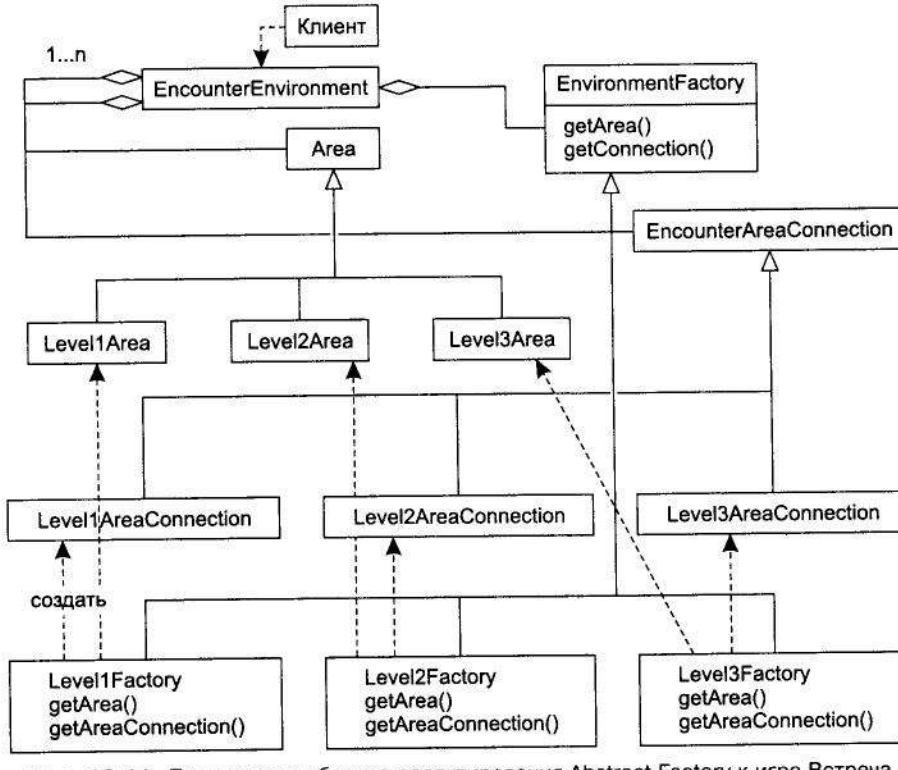


Рис. 10.11. Применение образца проектирования Abstract Factory к игре Встреча

Для перехода от старого проекта к новому нужен отдельный план. Этот план приведен на рис. 10.12. Он начинается с существующего проекта и состоит в добавлении и тестировании компонентов, не нарушающих имеющуюся реализацию. Перед последним шагом все готово к реализации образца проектирования Abstract Factory, причем все компоненты уже протестированы. На последнем шаге происходит окончательная реализация и тестирование.

10.4.4. Реализация запроса на сопровождение

Стадия реализации запросов на сопровождение описывается в табл. 10.5.

Таблица 10.5. Реализация запроса на сопровождение

	IEEE 1219-1992
	Стадия сопровождения 4: реализация
а. Входные данные	Первичный исходный код
	Первичная проектная документация
	Подробный проект с предыдущей стадии
б. Процесс	Внести в код необходимые изменения и дополнения
	Выполнить модульное тестирование
	Проверить готовность к системному тестированию

	IEEE 1219-1992
	Стадия сопровождения 4: реализация
в. Контроль	Проверка кода
	Верификация: контроль конфигурации и прослеживаемость нового кода
г. Выходные данные	Обновленные: программное обеспечение, отчеты о модульном тестировании, пользовательские документы
д. Выбранные факторы качества	Гибкость
	Прослеживаемость
	Понятность
	Удобство сопровождения
	Надежность
е. Выбранные метрики	Количество строк кода
	Процент ошибок

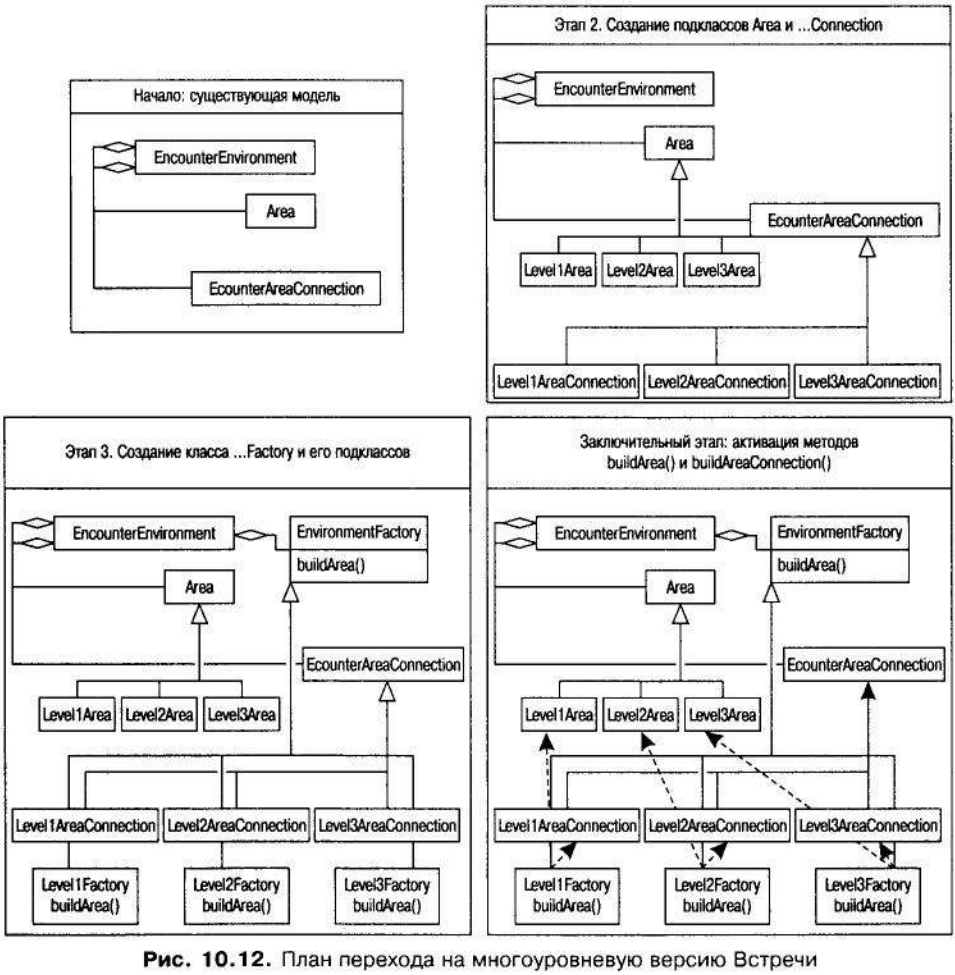


Рис. 10.12. План перехода на многоуровневую версию Встречи

Обработка запросов на сопровождение может потребовать разработки большого объема кода, в результате чего могут возникнуть новые дефекты. Процент ошибок — это количество дефектов, созданных в рамках трудозатрат по данной теме, в расчете на единицу измерения трудозатрат (например, человеко-месяц). Необходимо точно определить методику измерения количества новых дефектов. Можно, например, подсчитать «количество дефектов, найденных в течение трех месяцев с даты поставки». Предположим, что на обработку запроса 162 ушло 20 человеко-дней, за которые было создано 10 новых дефектов. Процент ошибок в этом случае будет составлять 10/20 = 0,5 дефекта/человеко-день.

Оставшиеся стадии процесса сопровождения — системное тестирование, приемка и обновление проектной документации — практически полностью аналогичны соответствующим фазам обычной разработки.

10.5. Управление сопровождением

План сопровождения (maintenance plan) регламентирует поток запросов на сопровождение внутри организации. Типичный план сопровождения приведен на рис. 10.13. Жирной линией на нем отмечена номинальная последовательность обработки запросов. В этом плане отдел сопровождения принимает от пользователей (заказчиков) жалобы и предложения. Они оформляются как запросы на сопровождение. Специальное подразделение, которое может быть как одним человеком, так и целым комитетом, принимает решение о реализации запросов и присваивает им приоритеты. Такой комитет иногда называется *Советом по контролю изменений* (CCB). После этого запросы обслуживаются техническим персоналом службы сопровождения. Тонкими линиями показаны другие последовательности появления и исчезновения запросов. Оптимальная организация работ по сопровождению зависит от масштабов приложения: процесс реализации запросов может быть достаточно растянутым.

С реализацией запросов связаны две проблемы. Первая — доставка подготовленного кода пользователям. Вторая — борьба с дефектами в целом. Дефект может повлиять на длительность тестирования, подготовка которого может недопустимо затянуться. Исключительным примером является военное приложение, в работах над которым довелось принять участие автору. Между определением задачи и полной реализацией запроса в этом проекте проходило до 9 месяцев! В таких ситуациях прибегают к выпуску *исправлений* или *заплат* (patch). Исправления — это изменения кода, которые либо устраняют дефект, либо позволяют обойти его. Исправления считаются временными. Часто они выпускаются в виде набора файлов, заменяющих уже написанный объектный код. Возможный вариант работы с исправлениями в организации, занимающейся разработкой, иллюстрирует рис. 10.14. Преимущества и недостатки исправлений рассматриваются в табл. 10.6.

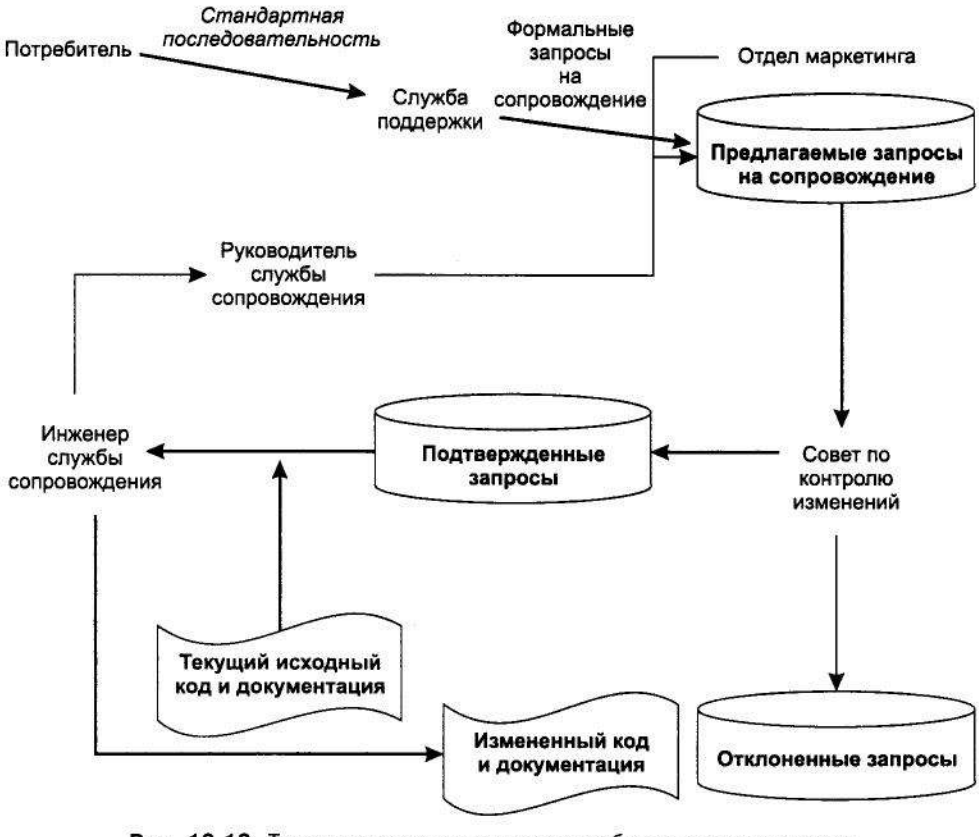


Рис. 10.13. Типичная последовательность работ по сопровождению

Таблица 10.6. Преимущества и недостатки исправлений

Преимущества	Недостатки
Быстрая удовлетворенность заказчиков	Дублирование работ: требуется реализация как временного, так и постоянного исправления
Возможность непрерывной работы и тестирования без широкого распространения дефектов	Иногда исправление остается навсегда, а нормальная версия так и не выходит
Исключает скрытие других дефектов	Затрудняется выпуск постоянного исправления, предназначенного для устранения временного
Позволяет тестировать исправление	Затрудняется процесс документирования

Под скрытием дефектов подразумевается то, что наличие неустраненного дефекта затрудняет обнаружение других дефектов, которые проявились бы, если бы первый дефект был устранен.

Рис. 10.14. Сопровождение и исправление

Опросы организаций, занимающихся сопровождением программ, показали, что эти организации ранжируют причины своих затруднений в следующем порядке [22].

1. Изменение приоритетов.
2. Методы тестирования.
3. Измерение производительности.
4. Неполная системная документация или ее отсутствие.
5. Адаптация к изменяющимся требованиям бизнеса.
6. Количество незавершенных задач.
7. Измерение вклада.
8. Низкий уровень самосознания из-за отсутствия признания или уважения.
9. Недостаток персонала, в особенности квалифицированного.
10. Недостаточный уровень методологии, стандартов, средств и процедур сопровождения.

Видно, что наибольшие трудности вызываются частым изменением приоритетов задач, стоящих перед службой сопровождения. В качестве примера рассмот-

рим нашу игру *Встреча*. Приведенная ниже последовательность изменения основных приоритетов типична для любой огранизации-разработчика:

- в момент выпуска — в игре должно быть меньше ошибок, чем у конкурентов:
  - решение: устранить максимальное количество дефектов;
- через два месяца после выпуска — у игры должно быть больше достоинств, чем у ее основного конкурента;
  - решение: быстрое выполнение усовершенствований;
- через шесть месяцев после выпуска — нужно снизить постоянно возрастающую стоимость сопровождения;
  - решение: устранение только самых серьезных дефектов.

Второй и третий по масштабу источники проблем связаны с методами тестирования и измерения производительности. Как мы уже видели, список возможных тестов достаточно широк, а время их выполнения весьма продолжительно. Измерять эффективность предприятия, занимающегося сопровождением, тоже можно множеством способов. Сопровождение часто оказывается одной из важных статей расходов, но при этом недооценивается. Еще один источник проблем — задержка выполнения запросов, которые накапливаются с опасной быстротой. Наконец, сопровождение и тестирование часто считаются менее престижными и эффектными профессиями, чем проектирование.

10.6. Качество сопровождения

Сопровождение нередко кажется тяжким грузом, но его можно рассматривать и как возможность продемонстрировать высокое качество обслуживания потребителей. Беннетт [9] пишет, что такое отношение к сопровождению широко распространено в Японии. Качественное сопровождение можно считать необходимым условием для достижения постоянной удовлетворенности покупателя и для получения его будущих заказов. Некоторые предприятия выделяют сопровождение в самостоятельный вид деятельности, потому что оно может стать надежным долгосрочным источником дохода.

10.6.1. Метрики сопровождения

Поскольку на сопровождение уходит значительная часть общей суммы трудозатрат за время жизни приложения (иногда для предприятий это оказывается неожиданностью), особенно важным становится наличие адекватной методики оценки затрат на сопровождение и доходов от него. Перечислим основные метрики сопровождения:

- количество строк сопровождаемого кода;
- трудозатраты на решение задач по сопровождению;
- количество дефектов.

Прежде всего следует определить цели сопровождения конкретного приложения, после чего выбрать дополнительные метрики, с помощью которых можно будет оценивать успех в достижении поставленных целей. Зависимость выбора метрик от поставленных целей демонстрирует табл. 10.7, построенная на основе аналогичной таблицы из [101].

Таблица 10.7. Выбор метрик в соответствии с целью сопровождения

Цель	Вопрос	Выбор метрики
Максимальное удовлетворение потребителя	Какие недостатки сказываются на удовлетворении потребителя?	* (1) частота отказов
		* (30) среднее время до отказа
		* отношение дефектов и исправлений ([Количество дефектов, появившихся в результате сопровождения]/[Количество устраненных дефектов])
	Сколько времени уходит на устранение недостатков?	* устранение отказов (среднее время на исправление недостатка с момента начала работ по исправлению)
		* длительность существования дефектов (среднее время от обнаружения дефекта до валидации исправления)
	Каковы основные препятствия?	* коэффициент использования персонала по видам задач (среднее количество человеко-месяцев на обнаружение и исправление каждого дефекта)
		* коэффициент использования компьютеров (среднее рабочее время / среднее системное время на один дефект)
Оптимизация трудозатрат и графика	На что уходят силы?	Трудозатраты и затраты календарного времени в расчете на дефект по категориям сложности
		* планирование
		* воспроизведение ситуации
		* отчет об ошибке
		* устранение недостатков
		* усовершенствование
Минимизация количества дефектов (продолжение сосредоточенного тестирования по схеме разработки)	Где наиболее вероятно обнаружение дефектов?	* (13) Количество точек входа и выхода для каждого модуля
		* (16) Цикломатическая сложность (см. раздел 7.6.1.2)

* Для метрик, взятых из стандарта IEEE, в таблице приведены соответствующие номера

Рассмотрим эти метрики более подробно.

(1) Частота отказов. Отказы — это дефекты, обнаруженные во время тестирования или в процессе эксплуатации приложения. Вычисляется как отношение количества найденных дефектов к величине NCSLOC. Последняя аббревиатура расшифровывается как «тысяч строк исходного кода не считая комментариев».

(30) Среднее время до отказа. Среднее время получения отказа после запуска приложения. Эта метрика требует введения определения отказа для тестируемого приложения. Определение зависит от того, что будет восприниматься потребителем как отказ. Отказом может считаться как аварийное завершение приложения, так и возникновение определенных конкретных проблем. Для финансового приложения как отказ может быть определена ошибка в вычислениях на величину не менее 1 доллара.

10.6.2. Применение метрик сопровождения

В этом разделе обсуждаются вопросы применения метрик для управления действиями по сопровождению. Доля комментариев в общем числе строк исходного кода позволяет предсказать масштаб трудозатрат на сопровождение (рис. 10.15). По сравнению с тремя другими модулями модуль «Запись неудачных дней» создаст больше всего трудностей при сопровождении из-за большой доли некомментированных строк и своего большого объема. Сопровождать модуль «Отчет о прибылях» будет проще всего, потому что он имеет наименьшие размеры и высокую долю комментариев. Долю комментариев можно вычислить при помощи специальной программы или путем изучения взятых наугад участков кода.

Рис. 10.15. Оценка трудозатрат на сопровождение

Для управления затратами на сопровождение полезны графики, аналогичные приведенному на рис. 10.16. В соответствии с этим графиком большое количество запросов на исправления и усовершенствования прибывает в первые два года, в результате чего на второй год эксплуатации возникает пик задержек выполнения запросов. Постепенно задолженность устраняется. Общий вид кривых на этом графике достаточно типичен, изменяется только масштаб по оси времени (годы, месяцы, недели).

Рис. 10.16. Профиль количества запросов на устранение недостатков

Ранее в этой главе мы подчеркивали разницу между исправлением недостатков и усовершенствованием приложения. Обычно руководитель отдела сопровождения старается учитывать затраты на эти два вида деятельности отдельно друг от друга, чтобы усовершенствования оплачивались заказчиком. Для достижения эффективного распределения объема работ полезны графики, подобные рис. 10.17. На графике показано среднее количество недель ожидания решения по запросу на сопровождение. Время отсчитывается с момента поступления первого отчета. Для исправлений средний срок составляет около одной недели, а для усовершенствований — около четырех недель.

Рис. 10.17. Пример профиля задержки до принятия решения о выполнении запроса

10.6.3. Удобство сопровождения

Оман [85] выделил основные параметры исходного кода, влияющие на удобство сопровождения приложения. Проведенное им разбиение исходного кода по типам показано на рис. 10.18. Автор изменил предложенное Оманом представление, чтобы сделать рисунок доступнее. Более полный вариант той же схемы приведен на рис. 10.19.

Например, чем лучше система разбита на модули, тем проще ее сопровождать (рис. 10.19, Исходный код ▸ Управляющая структура ▸ Система). Чем лучше данные инициализируются, тем проще их сопровождать (рис. 10.19, Исходный код ▸ Информационная структура ▸ Компонент). Читатель, несомненно, обратит внимание на то, что большинство перечисленных качеств уже рассматривались в этой книге с точки зрения качества проектирования и реализации.

Вспомните, что основным мотивом использования образцов проектирования является обеспечение удобства сопровождения приложений. Например, образец проектирования State позволяет с легкостью добавлять новые состояния, не изменяя функциональность имеющихся. К сожалению, усовершенствованные методы разработки систем обычно приводят к увеличению, а не к уменьшению затрат на сопровождение [23]. Судя по всему, это связано с тем, что хорошо разработанные приложения проще изменять, поэтому мы чаще прибегаем к их адаптации к новым условиям.

Рис. 10.18. Влияние параметров исходного кода на удобство сопровождения (1)

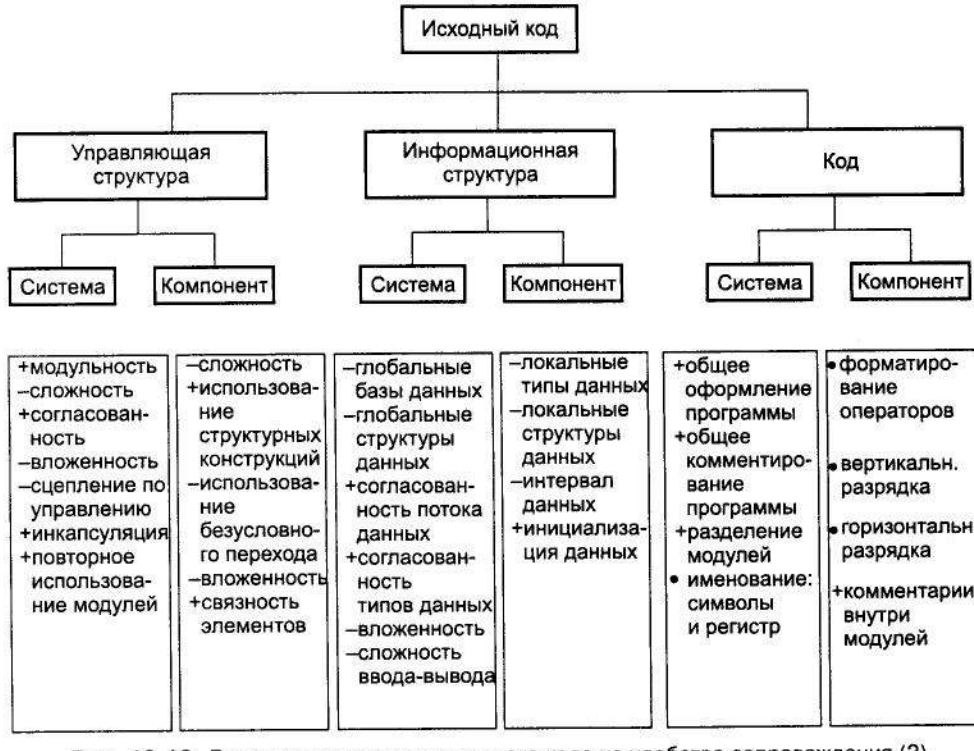


Рис. 10.19. Влияние параметров исходного кода на удобство сопровождения (2)

10.7. Подведение итогов

Сопровождение обычно поглощает значительную долю бюджета, выделяемого на разработку программ. В свою очередь, большую долю работ по сопровождению составляет усовершенствование приложения, а не устранение неполадок. Основные идеи главы приведены ниже.

- *Сопровождение программы* — это то, что следует за поставкой.
- Самое важное — это анализ влияния вносимых в приложение изменений.
- Стандарт IEEE описывает этот процесс:
  - определение, входные данные, процесс, контроль, выходные данные, качество, метрика;
  - порядок стадий тот же, что и при разработке.
- Сложности, возникающие при управлении:
  - управление потоком запросов;
  - создание мотивации у персонала;
  - обеспечение актуальности всей документации.
- Метрика: построение графиков для исправлений и усовершенствований.

Многими идеями и ссылками автор обязан Беннетту [8] и Пигоски [89]. Рекомендуем читателю также обратиться к полезному труду Омана [84].

Упражнения

Ответы и подсказки для упражнений, помеченных символами «о» или «п», приводятся в конце этой главы.

Вопросы для проверки

П10.1°. Дайте определение понятия «сопровождение программ» одним предложением.

П10.2°. Сопровождение бывает четырех видов, которые могут быть отнесены к двум категориям. Назовите их.

П10.3°. Приведите типичную последовательность обработки запросов на сопровождение. Решение вы найдете в разделе 10.2.

П10.4°. Предлагается изменить длину массива, используемого в некотором приложении, чтобы оно стало отвечать новым требованиям. Какие действия необходимо выполнить перед внесением изменений в код?

П10.5°. Определите *обратное проектирование* одним абзацем. Решение находится в разделе 10.3.2.

П10.6°. Приведите два или три способа использования унаследованных приложений в новых программах.

П10.7°. Что подразумевается под *реинжинирингом приложения*? Почему в этом может возникнуть необходимость?

П10.8°. Перечислите пять-десять возможных проблем, связанных с сопровождением (см. раздел 10.5).

Общие упражнения

О10.1. Предлагается реализовать приведенное ниже требование к игре *Встреча*, которое ранее считалась необязательным.

Требование к персонажу игрока: [желательно](«Внешний вид персонажа игрока») Игрок получает возможность выбирать изображение своего персонажа из четырех или более файлов формата GIF.

- ✦ К какому типу может быть отнесен этот запрос на сопровождение?
- ✦ Выполните оценку влияния выполнения этого запроса.

О10.2. Приведите примеры возможных корректирующих, адаптивных и упреждающих изменений для игры *Встреча*.

О10.3. В каких случаях из перечисленных далее может потребоваться реинжиниринг? Объясните.

- ✦ Преобразование модели банковских операций в автоматизированную систему безопасности банка.

- Расширение модели банковских операций с учетом перемещений персонала службы безопасности.
- Изменение электронной сетевой системы обучения таким образом, чтобы она могла в любое время обрабатывать тесты с вариантами выбора, использование которых позволяло бы студентам оценить понимание изучаемого материала.

Упражнения в команде

К10.1 (Сопровождение.)

1. Получить спецификации от двух других команд того же класса. Предложить по крайней мере одно изменение каждого типа: исправление, адаптация, усовершенствование и упреждение.
2. Другая команда должна выдвинуть аналогичные предложения к вашему проекту. Выполните оценку влияния предложенных изменений на ваш проект.
3. Обсудите предложенные изменения с выдвинувшей их командой на предмет доступных ресурсов. Реализуйте изменения.
4. Реализуйте, протестируйте и оцените свои изменения.

Критерии оценки.

1. Соответствие предложенных изменений указанному типу («Отлично» — точное соответствие).
2. Полнота оценки влияния («Отлично» — раскрыты все возможные аспекты влияния).
3. Качество тестирования изменений («Отлично» — все изменения тщательно протестированы).

Ответы

П10.1. Сопровождение программы — это ее обслуживание после поставки заказчику.

П10.2. Исправление (коррекция и адаптация) и усовершенствование (улучшение и упреждение).

П10.4. Оценка влияния — выявление артефактов, которые будут затронуты предлагаемым изменением.

П10.6. Одна из возможностей — изменение унаследованного приложения и добавление нового кода. Вторая возможность — вызов унаследованного приложения напрямую из нового. Третья возможность — создание обертки для унаследованного приложения (снабжение его адекватным интерфейсом) и использование его функциональности в новом приложении через этот интерфейс.

П10.7. Реинжиниринг существующего приложения — это его перепроектирование, масштаб которого превосходит мелкие изменения, но не достигает уровня

проектирования и реализации с нуля. Реинжиниринг рекомендуется применять в том случае, когда ожидаемое непрерывное изменение приложения приведет к ухудшению его как целого и к увеличению затрат на сопровождение.

Пример. Сопровождение игры Встреча

В этом примере рассматривается обработка запросов на сопровождение для игры *Встреча*.

Запрос на сопровождение 4593

(Содержимое настоящего документа описано в таблицах раздела 10.4.)

1. Определение задачи

1.1. Входные данные

Руководитель отдела маркетинга Мэри Краус определила, что механизм изменения значений характеристик двух встречающихся персонажей слишком сложен для среднего любителя компьютерных игр.

1.2. Процесс

Работа по этому запросу будет включена в выпуск 4. Запрос был принят к исполнению руководителем проекта 12 января 2000 года. Инженер Алан Оуэнс оценивает стоимость исправления в 12 человеко-часов (включая все необходимые тесты) при условии, что регрессионное тестирование для данного запроса будет объединено с регулярным регрессионным тестированием, которое проводится каждые две недели. Запросу установлен приоритет 4 (из 5, где 5 — низший возможный приоритет).

1.3. Контроль

Запрос был внесен в хранилище запросов инженером А. Джонсом 13 января 2000 года.

1.4. Выходные данные

13 января 2000 года инженер А. Джонс присвоил запросу номер 4593. Формулировка запроса такова.

Упростить формулу вычисления значений характеристик персонажей при их контакте друг с другом указанным ниже способом. Значения специфичных для зоны характеристик для более слабого персонажа уменьшаются вдвое. (Под специфичными для зоны характеристиками понимаются характеристики, относящиеся к зоне, в которой происходит контакт.) Значения характеристик более сильного персонажа не изменяются.

1.5. Факторы качества

Факторы качества — полнота и ясность запроса на сопровождение.

1.6. Метрики

Запрос 4593 был оценен в процессе проверки службы сопровождения 22 января 2000 года и получил следующие баллы:

- количество недочетов в запросе: 1;
- ясность запроса: 8 (0 — непонятен, 10 — невозможно сказать яснее);
- перекрытие запроса: 0 (0 — не перекрывается с другими запросами, 10 — включен в один или несколько незавершенных запросов);
- оценка задержки запроса: 12 дней.

[Примечание для студентов. Приведенные ниже сводные данные не следует повторять для каждого запроса.]

По всем незавершенным запросам получены следующие средние показатели:

- среднее количество недочетов в запросе: 0,9 (групповой стандарт — 0,5);
- средняя ясность запроса: 7,2 (групповой стандарт — 7);
- среднее перекрытие запроса: 1,6 (групповой стандарт — 2,0);
- средняя оценка задержки запроса: 18 дней (цель — 14).

2. Анализ задачи

2.1. Входные данные

1. Запрос на сопровождение 4593.
2. SRS версии 4.6.5 со следующими сведениями:

3.2.КО.3.1. Вступление в контакт с внешним персонажем [важно]. При встрече двух персонажей более сильным считается тот, у которого больше сумма значений характеристик, специфичных для зоны контакта. Система передает половину значения каждой характеристики, специфичной для зоны контакта, от более слабого персонажа к более сильному.

Предположим, что игрок встречается с внешним персонажем в зоне, где важны выносливость и сосредоточенность. Пусть p_c — значение выносливости игрока, и т. д. Предположим, что $p_c + p_к > f_c + f_к$. Тогда $p'_c = p_c + f_c/2$, $p'_к = p_к + f_к/2$. $f'_c = f_a/2$, $f'_к = f_к/2$, где x' — значение x после передачи характеристик.

Рассмотрим пример с конкретными числами. Если значение выносливости игрока равно 7, а сосредоточенности — 19, тогда как для Фредди (внешнего персонажа) выносливость равна 11, а сосредоточенность — 0,1, то игрок будет сильнее. В результате контакта характеристики персонажей получат следующие значения:

Игрок: выносливость 7 + 11/2 = 12,5; сосредоточенность 19 + 0,1/2 = 19,05 (отображается как 19,1).

Фредди: выносливость 11/2 = 5,5; сосредоточенность 0, потому что 0,1/2 меньше 0,1.

2.2. Процесс

Возможность осуществления: данный запрос считается осуществимым, поскольку его влияние ограничено результатами контакта и жизнью игровых персонажей.

Подробный анализ: единственный класс, который может быть изменен при выполнении запроса, — это *Контакт*. Объект *Контакт* вычисляет конечные значения характеристик персонажей, встретившихся друг с другом. Значения атрибутов объектов персонажей во время выполнения будут другими, но их код не требует изменения.

В описании запроса приводится преобразование $f_c' = f_a/2$, которое должно быть заменено на $f_c' = f_c/2$, потому что значение делится пополам, а не заменяется половиной другого значения.

Уточнение документации: необходимо изменить одно требование в SRS. Другие требования остаются нетронутыми. Необходимо изменить руководство пользователя.

2.3. Контроль

Запрос обсуждался на собрании отдела сопровождения 3 апреля 1999 года и был классифицирован как запрос уровня 1 (технически простой).

Для регрессионного тестирования были выбраны сегменты, задокументированные под номерами 7829 и 8924. Они не требуют изменений, за исключением таблиц ожидаемых результатов.

[Примечание для студентов. Документация по тестированию в настоящем примере отсутствует.]

2.4. Выходные данные

Одно требование в SRS было изменено так, как показано ниже (удаленные слова перечеркиваются, добавленные — подчеркиваются).

3.2.КО.3.1. Вступление в контакт с внешним персонажем [важно]. При встрече двух персонажей более сильным считается тот, у которого больше сумма значений характеристик, специфичных для зоны контакта. ~~Система передает половину значения каждой характеристики, специфичной длязоны контакта, от более слабого персонажа к более сильному.~~ Значения всех специфичных для зоны контакта характеристик более слабого персонажа уменьшаются вдвое.

Предположим, что игрок встречается с внешним персонажем в зоне, где важны выносливость и сосредоточенность. Пусть p_c — значение выносливости игрока, и т. д. Предположим, что $p_c+p_к > f_c+f_к$. Тогда ~~$p'_c=p_c+f_c/2$, $p'_к=p_к+f_к/2$, $f'_c=f_c/2$, $f'_к=f_к/2$~~ $f'_c=\lfloor f_c/2 \rfloor$ и $f'_к=\lfloor f_к/2 \rfloor$, где x′ — значение x после передачи характеристик. Выражение $\lfloor z \rfloor$ (операция округления вниз) означает наибольшее целое число, не превышающее z.

Рассмотрим пример с конкретными числами. Если значение выносливости игрока равно 7, а сосредоточенности — 19, тогда как для Фредди (внешнего персонажа) выносливость равна 11, а сосредоточенность — 0,1, то игрок в рассматриваемой зоне будет сильнее. В результате контакта характеристики персонажей получат следующие значения:

Игрок: ~~выносливость 7 + 11/2 = 12,5; сосредоточенность 19 + 0,1/2 = 19,05 (отображается как 19,1).~~ без изменений.

Фредди: выносливость $\lfloor 11/2 \rfloor = 5$; сосредоточенность $\lfloor 1/2 \rfloor = 0$.

В руководство пользователя необходимо внести следующие изменения... (не приводятся).

План реализации: данный запрос будет реализован инженером Т. Файном. Поскольку запрос приведет к значительным изменениям в стиле игры, он будет реализован и протестирован отдельно от всех прочих запросов.

Существует некоторая неопределенность относительно того, как это изменение будет принято игроками, поэтому планируется проведение расширенного тестирования удобства и простоты использования.

2.5. Факторы качества

Принципиальным фактором качества является цельность анализа влияния.

2.6. Метрики

Количество часов на анализ влияния одного измененного или добавленного метода: 0,5.

Степень изолированности от других запросов: 6,5 (абсолютно не изолирован — 0, полностью изолирован — 10). Примечание: этот запрос устранил дефект «$f'_c = f_a/2$» в первом требовании к контакту, в результате чего образовалось перекрытие с другим действием по сопровождению.

Ожидаемое время подтверждения задачи: 12 дней фактического времени.

Процент ошибок в документации: (еще не оценен).

Трудозатраты: 2 человеко-часа.

Фактическое время: 2 дня.

Процент ошибок: (еще не оценен).

В результате для полного набора незавершенных запросов получены следующие средние результаты.

- Среднее количество часов на анализ влияния одного измененного или добавленного метода: 0,9 (среднее по группе — 0,6).
- Средняя степень изолированности запросов друг от друга: 5,1 (групповой стандарт — 4,0).

3. Проектирование

3.1. Входные данные

Запрос на сопровождение 4593.

3.2. Процесс

Требуется изменение SDD. Псевдокод функции `execute()` класса *Контакт* был изменен так, как показано ниже.

3.3. Контроль

Требуется инспектирование нового псевдокода. Дата инспектирования — 8 апреля 1999 года.

3.4. Выходные данные

Номер первой редакции SDD с этим запросом — 5.4.3.

Приведенный ниже псевдокод должен быть включен в измененный метод execute() класса *Контакт*.

```
Пусть asq[] - специфичные для зоны контакта характеристики.
ПРИСВОИТЬ p сумму значений asq[] для игрока.
ПРИСВОИТЬ f сумму значений asq[] для внешнего персонажа.
ЕСЛИ p == f, контакт ничем не заканчивается.
ЕСЛИ p > f
        Уменьшить значения asq[] внешнего персонажа вдвое
        и применить к ним операцию округления вниз.
ИНАЧЕ ЕСЛИ f > p
        Уменьшить значения asq[] игрока вдвое и применить
        к ним операцию округления вниз.
```

План тестирования должен быть изменен следующим образом: ...

В связи с важностью данного запроса отдел маркетинга назначил тест удобства и простоты использования за номером 8902. Этот тест призван определить реакцию пользователей на вычисления по новым формулам.

3.5. Качество

Важнее всего то, как изменение будет принято игроками. Кроме того, важно проведение регрессионного тестирования, гарантирующего отсутствие ошибок в расчетах, а также отсутствие отрицательного влияния нововведения на другие функции игры.

3.6. Метрики

Количество замен в псевдокоде: 1.

Процент ошибок в документации: подлежит оценке.

Трудозатраты: 1 человеко-час.

Фактическое время: 1 день.

Процент ошибок: подлежит оценке.

4. Реализация

4.1. Входные данные

SDD версии 5.4.2.

Исходный код версии 11.3.7.

4.2. Процесс

Кодирование изменений по данному запросу было выполнено Б. Марксом. Несистемное тестирование произвел А. Антонини.

4.3. Контроль

Планы реализации и модульного тестирования были проверены 30 января 2000 года командой 5.

4.4. Выходные данные

Первая версия исходного кода с внесенными изменениями — 11.3.8.

Отчет об инспектировании находится в архивах инспектирований и датирован 30 января 2000.

Отчеты о тестировании находятся в пакетах документации тестирования 7820, 7621 и 8902.

4.5. Факторы качества

Применялись факторы качества SQAP проекта *Встреча*.

4.6. Метрики

...

5. Системное тестирование

5.1. Входные данные

SDD версии 5.4.2.

SRS версии 4.6.5.

Исходный код версии 11.3.8.

5.2. Процесс

Регрессионные тесты 7893.4, 23689.14, 21376.0 и 1237.46 пройдены. Новые регрессионные тесты, построенные для проверки новых расчетов, имеют номера 7893.5, 23689.15, 21376.1 и 1237.47 соответственно.

Перечисленные далее тесты будут проведены для версии 11.3.2:...

Удобство и простота использования конечного продукта будет оценено при помощи приведенной ниже процедуры, которая будет выполнена три раза. Полная документация тестирования сохранена за номером 89041.0.

Из всего множества игроков будут случайным образом выбраны 100 человек, которые должны будут оценить текущую версию игры по десятибалльной шкале по следующим показателям: интерес, сложность, реалистичность и общее удовольствие.

После этого им будет предоставлена измененная версия вместе с новым руководством пользователя. Они будут играть в измененную игру 10–15 часов в течение недели, после чего должны будут выполнить ту же оценку, что и ранее.

Приведенная выше процедура должна быть повторена с другой выборкой, но в обратном порядке (сначала новая версия, потом старая).

Отчет о результатах тестирования на удобство и простоту использования должен быть предоставлен в форме процентных различий между значениями перечисленных показателей для старой и новой версий.

5.3. Контроль

См. SCMP.

5.4. Выходные данные

Результаты тестирования будут сохранены в STD за номером 890451.

5.5. Факторы качества

См. STD.

5.6. Метрики

См. STD.

6. Приемосдаточное тестирование

Выполнение запроса на сопровождение будет признано успешным, если для 400 потребителей будут получены следующие результаты.

1. Пройдены все регрессионные тесты (см. выше).
2. По всем категориям достигнуто увеличение показателей.

По меньшей мере в двух категориях теста на удобство и простоту использования получено увеличение на 25 %. Ни по какой категории результат не может быть более чем на 3 % хуже, чем до внесения изменений.

Сокращения

ACM (Association for Computing Machinery) — Ассоциация вычислительной техники.

ANSI (American National Standards Institute) — Национальный институт стандартизации США.

API (application programming interface) — интерфейс прикладного программирования.

BPR (business process reengineering) — реинжиниринг бизнес-процесса.

CAD/CAM (Computer aided design / computer aided manufacturing) — автоматизированное проектирование / автоматизированное производство, САПР / АСУ ТП.

CASE (Computer-aided software engineering) — автоматизированная разработка программ.

CCB (Change control board) — Совет по контролю изменений.

CDR (critical design review) — критический обзор проектных решений.

CI (configurationitem) — элемент конфигурации.

CICS (customer information control system) — система управления информацией заказчика.

CM (configuration management) — управление конфигурациями.

CMM (Capability Maturity Model) — Модель зрелости возможностей.

COCOMO (Constructive Cost Model) — Конструктивная модель стоимости.

COM (Common object model, Microsoft) — Единая объектная модель.

CORBA (Common object request broker architecture) — Общая архитектура посредника запросов к объектам.

DBMS (Data base management system) — система управления базами данных, СУБД.

DFD (data flow diagram) — диаграмма потоков данных.

FP (function point) — единица функционального размера.

GUI (graphical user interface) — графический интерфейс пользователя.

IDE (interactive development environment) — интерактивная среда разработки.

IDL (Interface definition language) — язык определения интерфейса.

IEEE (Institute of Electrical, and Electronics Engineers) — Институт инженеров по электротехнике и радиоэлектронике.

IFPUG (International function point users group) — Международная группа пользователей функционального измерения.

IP (Inegtration plan) — План интеграции.

ISO (International Organization for Standardization) — Международная организация по стандартизации.

IV&V (independent verification and validation) — независимая экспертиза.

KLOC (kilo-lines of code) — тысяча строк программного кода.

LOC (lines of code) — число строк программного кода.

MR (maintenance request) — запрос на сопровождение.

MTBF (mean time between failures) — среднее время наработки на отказ.

MTTF (mean-time-to-failure) — средний период ошибки.

NCSLOC (non-comment lines of source code) — исполнимые строки исходного кода.

OMG (Object Management Group) — Консорциум по технологии манипулирования объектами.

OO (Object-oriented) — объектно-ориентированный, подход.

PCA (physical configuration audit) — аудит физической конфигурации.

PDR (preliminary design review) — предварительный обзор проектных решений.

PSD (Personal software documentation) — Индивидуальная программная документация.

PSP (Personal Software Process) — Индивидуальный процесс разработки программного обеспечения.

QA (quality assurance) — контроль качества.

RAD (Rapid application development) — быстрая разработка приложения.

RMI (Remote method invocation) — удаленный вызов метода.

RPG (role-playing game) — ролевая игра.

SCMP (Software configuration management plan) — План управления конфигурациями программного обеспечения.

SADT (Structured Analysis and Design Technique) — Технология структурного анализа и проектирования.

SDD (Software design document) — Проектная документация программного обеспечения.

SDF (Software development folder) — Рабочая папка проекта.

SEI (Software Engineering Institute) — Институт технологий разработки программного обеспечения.

SPMP (Software project management plan) — План управления программным проектом.

SQA (software quality assurance) — контроль качества программного обеспечения.

SQAP (Software quality assurance plan) — План контроля качества программного обеспечения.

SQL (Structured Query Language) — язык структурированных запросов.

SRS (Software requirements specification) — Спецификация требований к программному обеспечению.

STD Software test documentation — Документация по тестированию программного обеспечения.

STL (Standard Template Library) — библиотека стандартных шаблонов.

STP (Software test plan) — План тестирования программного обеспечения.

SUDP (Software user documentation plan) — План пользовательской документации.

SVVP (Software verification and validation plan) — План экспертизы программного обеспечения.

TSP (Team Software Process) — Командный процесс разработки программного обеспечения.

UML (Unified modeling language) — Унифицированный язык моделирования.

USDP (Unified Software Development Process) — Унифицированный процесс разработки программного обеспечения.

V&V (verification and validation) — верификация и валидация.

Словарь терминов

C-требования (C-requirements) — требования, зафиксированные в форме, наиболее удобной с точки зрения заказчика приложения, участвующие также в формировании требований для разработчиков.

D-требования (D-requirements) — требования разработчика. Требования, изложенные в той форме, которая наиболее удобна для того, чтобы разработчики могли от них отталкиваться. Также используются при формировании требований заказчика.

Автоматизированная разработка программ (Computer-aided software engineering, CASE) — процесс разработки программного обеспечения с применением согласованного набора средств разработки. Эти средства строго соответствуют различным фазам процесса разработки.

Автоматизированное проектирование / автоматизированное производство, САПР / АСУ ТП (Computer aided design / Computer aided manufacturing, CAD/CAM) — графические приложения, применяемые в проектировании и производстве электроники, строительных конструкций, машин и механизмов.

Альфа-версия (alpha release) — предварительная версия приложения, передаваемая особо доверенным представителям заказчика и (или) внутренним пользователям с целью обеспечения обратной связи.

Анализ требований (requirements analysis) — процесс получения законченного письменного утверждения, которое определяет, какими должны быть функциональность, внешний вид, производительность и поведение приложения.

Артефакт (artifact) — данные, исходный код или информация любого типа, которые сотрудник получает или использует в процессе разработки. В частности, используются в описании USDP.

Архитектура программного обеспечения (software architecture) — всеобъемлющий проект приложения, включающий в том числе и его структурную декомпозицию.

Ассоциация вычислительной техники (Association for Computing Machinery, ACM) — профессиональная ассоциация предприятий, ведущих деятельность в сфере компьютерных технологий, в частности занимающихся программным обеспечением.

Атрибут (attribute) — переменная класса в целом (в отличие от переменной внутри метода).

Аудит физической конфигурации (physical configuration audit, PCA) — систематическая проверка имеющихся в распоряжении физических артефактов проекта, таких как документация, исходный код, файлы, магнитные ленты и диски.

Бета-версия (beta release) — предварительная версия приложения, передаваемая избранным представителям заказчика с целью выявления дефектов и обеспечения обратной связи.

Быстрая разработка приложения (Rapid application development, RAD) — процесс ускоренной разработки приложения или его части. Ради сокращения времени разработки, как правило, жертвуют документацией, проектированием и расширяемостью приложения.

Валидация (validation) — процесс, заключающийся в проверке того, что приложение выполняет свои функции именно так, как было задумано.

Вариант использования (use case) — последовательность действий, выполняемых как приложением, так и пользователем, типичная для данного приложения. При этом пользователю назначается некоторая роль, и применительно к этому варианту использования его называют *действующим лицом* (*актером*).

Верификация (verification) — процесс проверки того, что приложение строится в строгом соответствии с тем, как это запланировано.

Водопадный тип процесса разработки (waterfall) — процесс разработки программного обеспечения, состоящий из следующих фаз: сбор требований, проектирование, реализация в коде и тестирование. Все фазы выполняются последовательно, возможно лишь небольшое наложение соседних фаз.

Встреча (Encounter) — учебный пример видеоигры, изучаемый в этой книге.

Графический интерфейс пользователя (graphical user interface, GUI) — графический экран (часто интерактивный), посредством которого пользователь взаимодействует с приложением.

Действующее лицо (actor) — индивидуальная роль, присваиваемая пользователю приложения применительно к варианту использования.

Диаграмма последовательности (sequence diagram) — диаграмма, на которой изображены объекты приложения, отображающая последовательность обращений к функциям объектов. Обычно с помощью диаграмм последовательности уточняют варианты использования.

Диаграмма потоков данных (Data flow diagram, DFD) — диаграмма, отражающая потоки данных в направлении к приложению, внутри приложения и за его пределами. Данные перемещаются между пользователями приложения, хранилищами данных и элементами внутренней обработки в приложении.

Доказуемо корректная программа (provably correct program) — программа, написанная таким образом, что мы можем привлечь математический или логический аппарат для доказательства того, что она удовлетворяет предъявленным требованиям.

Документация по тестированию программного обеспечения (Software test documentation, STD) — документ, регламентирующий тестирование приложения во всех его аспектах.

Единица функционального размера (function point, FP) — мера сложности приложения.

Заинтересованные лица (stakeholders) — все те, кто так или иначе заинтересован в реализации проекта: заказчики и пользователи (которые сами могут и не

являться заказчиками), а также инвесторы, пользовательские группы и сами разработчики.

Запрос на сопровождение (maintenance request, MR) — запрос на модификацию или расширение возможностей существующего приложения.

Инкрементальный тип процесса разработки (incremental) — процесс разработки программного обеспечения, в котором число итераций так велико, что каждая следующая итерация предоставляет слишком мало новых возможностей по сравнению с предыдущей.

Инспектирование (inspection) — проверка частей проекта, таких как требования, результаты проектирования, программный код, на наличие дефектов. Обычно инспектирование проводит группа коллег автора работы.

Инвариант (invariant) — утверждение, устанавливающее связь между переменными в определенном контексте, которое сохраняется неизменным, хотя значения этих переменных могут меняться.

Индивидуальная программная документация (Personal software documentation, PSD) — документация, поддерживаемая каждым разработчиком индивидуально, в которой отражается текущее состояние его части кода.

Индивидуальный процесс разработки программного обеспечения (Personal Software Process, PSP) — процесс, разработанный Хэмфри (Институт разработки программного обеспечения) с целью оценки и улучшения производительности отдельно взятого разработчика программного обеспечения.

Институт инженеров по электротехнике и радиоэлектронике (Institute of Electrical, and Electronics Engineers, IEEE) — профессиональная организация, в центре внимания которой лежат разработки в области электроники, электротехники и программного обеспечения.

Институт технологий разработки программного обеспечения (Software Engineering Institute, SEI) — институт, созданный с целью обеспечить качество программного обеспечения, разрабатываемого для вооруженных сил США. Его достижения нашли применение в многочисленных организациях, не имеющих отношения к оборонной промышленности.

Интегральное тестирование (integration testing) — процесс тестирования, в ходе которого проверяется, насколько успешно объединены модули.

Интеграция (integration) — слияние отдельных модулей в одном приложении.

Интерактивная среда разработки (interactive development environment, IDE) — программное приложение, облегчающее разработчику создание, редактирование, компиляцию и исполнение кода.

Интерфейс (interface) — под интерфейсом системы понимается спецификация набора ее функций. Эта спецификация содержит имена функций, а также типы параметров, возвращаемых значений и исключений.

Интерфейс прикладного программирования (Application Programming Interface, API) — набор классов и прототипов член-функций, полезных для программистов. Содержит такие данные о функции, как ее имя, типы параметров, типы возвращаемых значений и генерируемые исключения.

Исполнимые строки исходного кода (non-comment lines of source code, NCSLOC) — число строк программного кода без учета комментариев.

Итерация (iteration) — повторяющийся процесс добавления требований, проектирования, разработки и тестирования с целью получения очередной сборки приложения.

Каркас (framework) — набор обобщенных классов, формирующий базис для нескольких приложений. Классы каждого из этих приложений получены из классов каркаса путем агрегации или наследования.

Командный процесс разработки программного обеспечения (Team Software Process, TSP) — процесс, разработанный Хэмфри (Институт разработки программного обеспечения) с целью оценки и улучшения производительности команд разработчиков.

Консорциум по технологии манипулирования объектами (Object Management Group, OMG) — некоммерческая ассоциация компаний, выпускающая стандарты в области распределенных вычислений.

Конструктивная модель стоимости (Constructive Cost Model, COCOMO) — формулы Боэма, позволяющие на основе оценки количества строк кода рассчитать предположительный объем трудозатрат (в человеко-часах), необходимый для построения приложения, и длительность проекта.

Контроль качества (quality assurance, QA) — процесс, с помощью которого можно обеспечить достижение заявленного уровня качества во время проектирования. Чаще используется для обозначения организации, выполняющей эту функцию.

Критический обзор проектных решений (critical design review, CDR) — процесс, в ходе которого принимается окончательное решение о том, приступать ли к предлагаемому проектированию.

Метод «белого ящика» (white box process) — метод (обычно метод тестирования), применяемый к реализации, который учитывает способ обработки данных, используемый в исследуемом коде.

Метод «черного ящика» (black box method) — метод (обычно метод тестирования), применяемый к реализации, который базируется только на входных и выходных данных (то есть не принимает в расчет способ обработки данных, используемый в исследуемом коде).

Метрика (metric) — количественная мера артефакта программного обеспечения. Например, *число строк кода* — это метрика исходного кода.

Модель (model) — модель приложения — это представление его проекта в определенном аспекте, таком как, например, множество его классов или поведение, управляемое событиями.

Модель зрелости возможностей (Capability Maturity Model, CMM) — системный метод, позволяющий оценить потенциал организации в целом относительно разработки программного обеспечения. Создан в Институте разработки программного обеспечения (Питтсбург).

Модульное (или компонентное) тестирование (unit testing) — тестирование части приложения в изоляции от остальных его частей.

Независимая экспертиза (independent verification and validation, IV&V) — процесс верификации и валидации, осуществляемый третьими лицами (то есть не организацией-разработчиком).

Нефункциональное требование (non-functional requirement) — требование, которое предъявляется к приложению, но никак не влияет на его функциональность. Примером может служить ограничение на объем памяти.

Образец проектирования (design pattern) — образец часто используемых классов, отношений между ними и сопутствующих им алгоритмов.

Обратное проектирование (reverse engineering) — построение содержимого фазы процесса разработки из артефактов последующей фазы, например получение модели из кода программы.

Общая архитектура посредника запросов к объектам (Common object request broker architecture, CORBA) — стандарт, в рамках которого приложения имеют возможность вызывать функции, размещенные на удаленных платформах, независимо от языка их реализации.

Объектно-ориентированный подход (object-oriented, OO) — организация проекта и кода по классам и экземплярам (объектам). Каждый объект класса обеспечен набором функций, определенных для этого класса, и каждый объект имеет копию определенного для этого класса набора переменных.

Обратное требование (inverse requirement) — спецификация требований, которые определяют, чего приложение не будет делать.

Оценка возможностей (capability assessment) — процесс, при помощи которого получают объективную количественную оценку возможностей организации, группы или отдельного разработчика.

Переход (в диаграмме переходов состояний) (transition) — процесс, в ходе которого объект меняет состояние с одного на другое.

План тестирования программного обеспечения (Software test plan, STP) — в этом документе указывается, какие части приложения должны быть протестированы, и приводится график тестирования.

План управления конфигурациями программного обеспечения (Software configuration management plan, SCMP) — документ, в котором устанавливается порядок управления кодом и документацией проекта, а также всех его версий.

План управления программным проектом (Software project management plan, SPMP) — план, определяющий, кто и в каком порядке какие части проекта будет разрабатывать.

Предварительный обзор проектных решений (preliminary design review, PDR) — совещание с участием разработчиков и менеджеров, на котором выносится на обсуждение эскизный проект (в целом или его часть).

Пригодный к прослеживанию (traceable) — требование является прослеживаемым, если фрагменты проекта и программного кода, в которых оно реализуется, могут быть легко индентифицированы. Требование не является прослеживаемым, если не ясно, какие элементы проекта выражают его и в какой части кода оно реализовано.

Пригодный к тестированию (testable) — артефакт, например требование, для которого имеется возможность написать детальный тест, проверяющий соответствие между продуктом и артефактом.

Прототип (prototype) — приложение, на котором можно проиллюстрировать или продемонстрировать некоторые аспекты разрабатываемого приложения.

Процесс (process) — программный процесс — это последовательность действий, из которых складывается разработка.

Псевдокод (pseudocode) — язык, подобный языку человеческого общения, и вместе с тем достаточно формальный для того, чтобы на нем было можно описывать алгоритмы.

Рабочая папка проекта (Software development folder, SDF) — документ, фиксирующий текущее состояние кода, над которым работает отдельный разработчик. В нем содержатся подробные результаты модульного тестирования, выполненного этим разработчиком на данный момент.

Регрессионное тестирование (regression testing) — процесс проверки того факта, что добавление кода в приложение, находящееся в разработке, не ухудшит тех возможностей, которыми оно на данный момент обладает.

Реинжиниринг бизнес-процесса (business process reengineering, BPR) — системный проект бизнес-процесса, например процесса управления закупками, от начальной стадии до завершения, включающий человеческие и другие аспекты. Обычно выполняется «с нуля».

Ролевая игра (role-playing game, RPG) — игра, в которой взаимодействие игроков осуществляется на основе распределенных между ними ролей.

Сборка (build) — частичная реализация приложения.

Система управления базами данных, СУБД (data base management system, DBMS) — система, управляющая организацией данных и доступом к ним.

Системная разработка (system engineering) — процесс анализа и разработки системы в целом, включая как аппаратное, так и программное обеспечение.

Системное тестирование (system testing) — тестирование приложения в целом.

Служба помощи (Help desk) — средство обеспечения помощи пользователям приложения.

Событие (event) — внешнее по отношению к объекту явление, оказывающее на него влияние.

Совет по контролю изменений (Change control board, CCB) — группа, принимающая решения о целесообразности реализации тех или иных расширений или изменений в приложении.

Сопровождение (maintenance) — процесс изменения приложения после поставки с целью исправления ошибок, повышения производительности или для адаптации к изменившимся условиям.

Состояние (state) — статус объекта. Формально определяется через набор значений переменных объекта. Например, можно сказать, что объект *Автомобиль* находится в состоянии *Раритетный*, если год его выпуска не больше 1955 и переменная, отвечающая за его внешний вид, имеет значение не хуже, чем *удовлетворительный*.

Спецификация требований к программному обеспечению (Software requirements specification, SRS) — в этом документе определяется, что должно делать приложение.

Спиральный тип процесса разработки (spiral) — процесс разработки программного обеспечения, суть которого заключается в том, что на каждой итерации строится очередная версия приложения на основе ее предыдущей версии.

Среднее время наработки на отказ (mean time between failures, MTBF) – вычисляется как отношение срока эксплуатации приложения к числу уникальных отказов, обнаруженных за это время. Для этого необходимо сформулировать понятие «отказ».

Средний период ошибки (mean-time-to-failure, MTTF) – измеряется путем запоминания промежутков времени между всеми парами замеченных последовательных ошибок и их усреднения.

Схема (roadmap) – список действий, приводящий к достижению поставленной цели.

Унаследованное приложение (legacy application) – приложение, которое уже поставляется заказчику и находится в эксплуатации.

Унифицированный процесс разработки программного обеспечения (Unified Software Development Process, USDP) – процесс разработки, созданный Якобсоном, Бучем и Рамбо, в котором основное внимание уделяется вариантам использования.

Унифицированный язык моделирования (Unified modeling language, UML) – графическая нотация, используемая для описания объектно-ориентированного проектирования.

Управление конфигурациями (configuration management, CM) – процесс, регламентирующий управление версиями различных артефактов программного проекта, а также их сопровождение.

Управление проектом (project management) – процесс исполнения обязанностей с целью успешного завершения проекта.

Устранение риска (risk retirement) – процесс ликвидации факторов, угрожающих успешному исполнению проекта. Можно или найти способ избежать риска, или принять меры по снижению его влияния на проект.

Формальные методы (formal methods) – строгие методы спецификации требований, проекта или реализации, базирующиеся на законах математики или логики.

Функциональное требование (functional requirement) – требование, выражающее одну из функций, которые должно выполнять приложение.

Элемент конфигурации (configuration item, CI) – артефакт, версии которого тщательно отслеживаются от начальной стадии проекта до его завершения.

Литература

1. Accreditation Board for Engineering and Technology (ABET), 1996.
2. Albrecht, A. J., «Measuring Application Development Productivity», Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium, October 1979, pp. 83–92.
3. Alspaugh, Thomas A.; Faulk, Stuart R.; Heninger Britton, Kathryn; Parker, R. Alan; Parnas, David L., and Shore, John E., «Software Requirements for the A-7E Aircraft», NRL Memorandum Report No. 3876, August 31, 1992.
4. Ambler, Scott, www.ambysoft.com (1999).
5. Anderson, A. et al., «At Chrysler, Objects Pay», Distributed Computing, October 1998, pp. 25–28.
6. «How Do You Define Software Architecture?», http://www.sei.cmu.edu/architecture/definitions.html (12/99).
7. Beck, Kent, «Extreme Programming: A Humanistic Discipline of Software Development», FASE, Lisbon, Portugal, part of ETAPS 1998, 1998.
8. Bennett, K. «Software Maintenance: A Tutorial», Software Engineering, IEEE Computer Society, November, 1999.
9. Bennett, K.H., «Software Maintenance in Japan» keynote paper, in Proc. 8th European Workshop on Software Maintenance, Durham, North Carolina, 1994.
10. Berry, Daniel M., and Lawrence, Brian, «Requirements Engineering», IEEE Software, vol. 15, no. 2, March/April 1998.
11. Boehm, Barry, Software Engineering Economics, Englewood Cliffs, NJ: Prentice Hall, 1981.
12. Bohm, C, and Jacopini, G., «Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules», Communications of the ACM, May 1966, pp. 366–371.
13. Booch, G., Object Solutions: Managing the Object-Oriented Project, New York: Addison-Wesley, 1995.
14. Booch, G., Object-Oriented Analysis and Design with Applications, Reading, MA: Addison-Wesley, 1994.
15. Booch, G.; Jacobson, I.; and Rumbaugh, J., The Unified Modeling Language User Guide, Reading, MA: Addison-Wesley, 1998.
16. Boston Globe, October 1, 1999.

17. Brackett, J., ftp://ftp.sei.cmu.edu/pub/education/cml9.ps, (January 1990).
18. Brooks, Frederick P., Jr., The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, Addison-Wesley, July 1995.

    Брукс, Ф. Мифический человеко-месяц. Как создаются программные системы. — СПб.: Символ-Плюс, 2000.
19. «The Mother of All Software Projects». Business Week, February 22, 1999.
20. http://www.egroups.com (4/2000).
21. Cucumano, A. and Selby, R. W., «How Microsoft Builds Software», Communications of the ACM, vol. 40, no. 6, June 1997, pp. 53–61.
22. Dekleva, S., «Delphi Study of Software Maintenance Problems», Proceedings of the IEEE Conference on Software Maintenance, Orlando, FL, November 1992, pp. 10–17.
23. Dekleva, Sasa M., «The Influence of the Information Systems Development Approach on Maintenance», MIS Quarterly, vol. 16, no. 3, pp. 355–372.
24. DeMarco, Tom, and Lister, Timothy, Peopleware: Productive Projects and Teams, 2nd edition, New York, NY: Dorset House, 1999.
25. Dijkstra, E., A Discipline of Programming, Englewood Cliffs, NJ: Prentice Hall, 1976.
26. Dijkstra, E.W., «GOTO statement considered harmful», letter in Communications of the ACM, March 1968.
27. Dorfman, M. (ed.) and Thayer, Richard H. (contributor), Software Engineering, IEEE Computer Society, November 1999.
28. Dreger, J., Function Point Analysis, Englewood Cliffs, NJ: Prentice Hall, 1989.
29. Engelmore, Robert, and Morgan, Tony (ed.), Blackboard Systems (The Insight Series in Artificial Intelligence), ASIN: 0201174316.
30. Fagin, M., «Design and Code Inspections to Reduce Errors in Program Development», IBM Systems Journal, vol. 15, no. 3, pp.182–211.
31. Foster, J. «Cost Factors in Software Maintenance», Ph. D. Thesis, University of Durham, NC, Computer Science Department, 1994, as noted in [Be2].
32. Fowler, Martin, Refadoring: Improving the Design of Existing Code, Reading, MA: Addison-Wesley, 1999.
33. Gamma, Erich; Helm, Richard; Johnson, Ralph; and Vlissides, John, Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing), Addison-Wesley, August 1999.
34. Garlan, David, and Shaw, Mary, Software Architecture: Perspectives on an Emerging Discipline, Englewood Cliffs, NJ: Prentice Hall, 1996.
35. Galitz, W, The Essential Guide to User Interface Design: An Introduction to GUI Principles and Techniques, New York: John Wiley & Sons, 1996.
36. Gehani, Narain, and Lally, S., Software Specification Techniques (International Computer Science), Reading, MA: Addison-Wesley, 1985.

37. Gilb, T., and Graham, D., Software Inspection, Reading, MA: Addison-Wesley, 1993.
38. Glass, R., «Maintenance: Less is Not More», IEEE Software, July/August 1998.
39. Gres, David, «The Science of Programming», New York: Springer-Verlag, 1987.
40. Hammer, Michael, and Champy, James, Reengineering the Corporation : A Manifesto for Business Revolution, Reprint edition, Harperbusiness, May 1994.
41. Harel, D., «On Visual Formalisms», Communications of the ACM, May 1988, pp. 514–530.
42. Hayes, I., Specification Case Studies, Englewood Cliffs, NJ: Prentice Hall, 1986.
43. Hayes-Roth, Frederick, Building Expert Systems, ASIN: 0201106868.
44. Hoare, C. A. R., and Jones, C.B., Essays in Computing Science (Prentice Hall International Series in Computer Science), Englewood Cliffs, NJ: Prentice Hall, 1989, ASIN: 0132840278.
45. Horstmann, Cay S., and Cornell, Gary, Core Java 2 , Volume 1: Fundamentals, Upper Saddle River, NJ: Prentice Hall, 1998.
46. Horstmann, Cay S., and Cornell, Gary, Core Java 1.1, Volume II: Advanced Features, Upper Saddle River, NJ: Prentice Hall, 1998.
47. Horstmann, Cay S., Practical Object-Oriented Development in C++ and Java, New York: John Wiley & Sons, 1997.
48. Humphrey, Watts S., A Discipline for Software Engineering (SEI Series in Software Engineering), Reading, MA: Addison-Wesley, 1995.
49. Humphrey, Watts S., Managing the Software Process (SEI Series in Software Engineering), Reading, MA: Addison-Wesley, 1989.
50. http://www.sei.cmu.edu/publications/articles/sources/practice.preach/index.html (12/99).
51. http://www.stsc.hill.af.mil/crosstalk/1998/apr/dimensions.html (12/99).
52. Humphrey, Watts S., Introduction to the Personal Software Process (SEI Series in Software Engineering), Reading, MA: Addison-Wesley, December 1996.
53. http://csg.uwaterloo.ca/~huang/dptool/interfac.htm (12/99).
54. Humphrey, Watts S., Managing Technical People: Innovation, Teamwork, and the Software Process (Sei Series in Software Engineering), Reading, MA: Addison-Wesley, 1996.
55. Humphrey, Watts S., Introduction to the Team Software Process (The SEI Series in Software Engineering), Addison-Wesley, August 1999.
56. IEEE Software Engineering Standards Collection, 1997 Edition, Piscataway, NJ: IEEE, 1997.
57. IEEE Standard Glossary of Software Engineering Terminology, IEEE 610.12-1990.
58. Software Test Documentation, ANSI/IEEE 829We-1983 (reaffirmed 1991).
59. Guide to the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, ANSI/IEEE 982.2-1988.

60. IEEE Std 1219-1998 (Revision of IEEE Std 1219-1992) IEEE Std 1219-1998 IEEE Standard for Software Maintenance, IEEE Computer Society.

61. International Function Point Users' Group, http://www.ifpug.org/ (12/99).

62. International Function Point Users' Group reference to function point spreadsheets, http://www.ifpug.org/home/docs/freebies.html (as of 12/99).

63. Jacobson, Ivar, Object-Oriented Software Engineering: A Use Case Driven Approach (Addison-Wesley Object Technology Series), Reading, MA: Addison-Wesley, 1994.

64. Jacobson, Ivar; Rumbaugh, James; and Booch, Grady, The Unified Software Development Process (Addison-Wesley Object Technology Series), Reading, MA: Addison-Wesley, 1999.

Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения. — СПб.: Питер, 2003.

65. Jacobson, Ivar; Griss, Martin; and Jonsson, Patrik, Software Reuse: Architecture Process and Organization for Business Success, Reading, MA: Addison-Wesley, 1997.

66. Jagannathan, V; Dodhiawala, Rajendra; and Baum, Lawrence S. (ed.), Blackboard Architectures and Applications (Perspectives in Artificial Intelligence, Vol. 3), ASIN: 0123799406.

67. http://www.java.sun.com/products/jdk/javadoc/index.html (12/99).

68. Jones, Capers, and Jones, Capers T., Applied Software Measurement: Assuring Productivity and Quality, 2nd edition, New York: McGraw-Hill, 1996.

69. Jordan, Richard; Smilan, Ruth; and Wilkinson, Alex, «Streamlining the Project Cycle with Object-Oriented Requirements», OOPSLA Conference Proceedings, 1994, pp. 287–300.

70. Jones, C., Software Development: A Rigorous Approach, Englewood Cliffs, NJ: Prentice Hall, 1980.

71. http://www.spr.com/library/01angtbl.htm (12/99).

72. Kaluzniacky, Eugene; Kanabar, Vijay; and Irwin, Richard D., Xbase Programming for the True Beginner: An Introduction to the Xbase Language in the Context of dBASE Iii+, Iv, 5, Foxpro, and Clipper, Burr Ridge, IL: Irwin, 1995.

73. Keil, M.; Cule, P.; Lyytinen, K.; and Schmidt, R., «A Framework for Identifying Software Project Risks», Communications of the ACM, vol. 41, no. 11, November 1998.

74. Kit, Edward, Software Testing in the Real World: Improving the Process, Reading, MA: Addison-Wesley, 1995.

75. Kruchten, P., The Rational Unified Process, Reading, MA: Addison-Wesley, 1998.

76. Lea, Doug, Concurrent Programming in Java: Design Principles and Patterns (Java Series), Reading, MA: Addison-Wesley, 1996.

77. Lehman, M., «Programs, Life Cycles, and the Laws of Software Evolution», Proc. IEEE, vol. 19, 1980, pp. 1060–1076.

78. Lehman, M., «Program Evolution Information Processing Management», Proc. IEEE, vol. 20, 1984, pp. 19–36.

79. Lientz, Bennet P.; Swanson, E. Burton; and Tompkins, G. E., «Characteristics of Applications Software Maintenance», Communications of the ACM (CACM), vol. 21, 1978, pp. 466–471.

80. Meyer, Bertrand, «On to Components», Computer IEEE, January 1999.

81. Musser, David R.; Saini, Atul (Contributor); Stepanov, Alexander; STL Tutorial & Reference Guide: C++ Programming With the Standard Template Library, (Addison-Wesley Professional Computing Series), Reading, MA: Addison-Wesley, 1996.

82. Myers, Glenford J., The Art of Software Testing, New York, John Wiley & Sons, 1979.

83. http://www-isds.jpl.nasa.gov/cwo/cwo_23/handbook/Dsnswdhb.htm (12/99).

84. Oman, P.; Hagemeister, J.; and Ash, D., «A Definition and Taxonomy for Software Maintainability», University of Idaho, Moscow, Software Engineering Test Laboratory Report #91-08-TR, ID 83843, 1992.

85. Oman, P., «HP_MAS: A Tool for Software Maintainability Assessment», University of Idaho, Moscow, Software Engineering Test Laboratory Technical Report #92-07-TR.

86. The Object Management Group, http://www.omg.org (12/99).

87. Ostrolenk, Gary; Tobin, Mary; and Southworth, Mark; «Lessons from Using Z to Specify a Software Tool», IEEE Transactions on Software Engineering, vol. 24, no. 1, January 1998.

88. Parnas, D. L., and Madey, J., «Functional Documentation for Computer Systems Engineering (Version 2)», McMaster University, Hamilton ONT, Canada, Technical Report CRL Report No. 237, September 1991.

89. Pigoski, Thomas M., Practical Software Maintenance: Best Practices for Managing Your Software Investment, New York: John Wiley & Sons, 1996.

90. http://www.rational.com (12/99).

91. Rand, Paul, A Designer's Art, New Haven, CT: Yale University Press, 1985.

92. Ross, Douglas T., «Structured Analysis (SA): A Language for Communicating Ideas», IEEE Transactions on Software Engineering, vol. 3, no. 1, January 1977, pp. 16–33.

93. Software Specification: A Framework, www.sei.cmu.edu/topics/publications/documents/cms/cm.011.html (12/99).

94. Rugaber, Spencer, and White, Jim, «Restoring a Legacy: Lessons Learned», IEEE Software, vol. 15, no. 4, July/August 1998.

95. Rumbaugh, James; Jacobson, Ivar; and Booch, Grady, The Unified Modeling Language Reference Manual, Reading, MA: Addison-Wesley, December 1998.

96. Rumbaugh, J.; Blaha, M; Premerlani, W; Eddy, F.; and Lorenson, W, Object-Oriented Modeling and Design, Englewood Cliffs, NJ: Prentice Hall, 1990.

97. Schwartz, J. T.; Dewar, R. B. K.; Dubinsky, E.; and Schonberg, E., Programming with Sets: An Introduction to SETL, New York: Springer-Verlag, 1986.
98. Sheldon, F. et al., «Reliability Measurement from Theory to Practice», IEEE Software, vol. 9, no. 4, July 1992.
99. Shlaer, Sally, and Mellor, Stephen J., Object Lifecycles: Modeling the World in States, Englewood Cliffs, NJ: Yourdon, 1991.
100. Shnayerson, Michael, The Car That Could; The Inside Story of General Motors' Revolutionary Electric Vehicle, ASIN: 067942105X.
101. Stark, George E., and Kern, Louise C., «A Software Metric Set for Program Maintenance Management», Journal of Systems and Software, vol. 24, 1994, pp. 239–249. http://www.members.aol.com/GEShome/met_prog/sustaining/estSus.html (12/99).
102. JavaTM Platform 1.1 Core API Specification http://www.java.sun.eom/products/jdk/l.1/docs/api/packages.html (12/99).
103. http://www.spr.com/library/01angtbl.htm (12/99).
104. http://www.suntest.com/ (8/99).
105. Szyperski, Clemens, Component Software: Beyond Object-Oriented Programming, Reading, MA: Addison-Wesley, 1998.
106. Teichroew, Daniel, and Hershey, Ernest A. Ill, «PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems», IEEE Transactions on Software Engineering (TSE), vol. 3, no. 1, January 1977, pp. 41–48.
107. Thayer, Richard H., Software Engineering Project Management, Second Edition, Piscataway, NJ: IEEE, 1997.
108. Ward, Paul T., and Mellor, Stephen J., Structured Development for Real-Time Systems: Essential Modeling Techniques, Englewood Cliffs, NJ: Yourdon, 1986.
109. Wegner, Peter, «Why Interaction Is More Powerful Than Algorithms», Communications of the ACM, May 1997, pp. 80–91.
110. Weiss, D. M., «Evaluating Software Development by Analysis of Change Data», SEL-81-011, NASA Software Engineering Laboratory, November 1981.
111. Wilde, Norman, «Software Impact Analysis: Process and Issues», SERC-TR-70-F, March 1994. Software Engineering Research Center, National Science Foundation, http://www.hesperus.oboe.com/serc/ (12/99) and Durham University Technical Report, July 7, 1993.
112. Yourdon, E., Decline and Fall of the American Programmer, Englewood Cliffs, NJ: Prentice Hall, 1992.
113. http://www.comlab.ox.ac.uk/archive/z.html (12/99).
114. Zobel, Justin, Writing for Computer Science: The Art of Effective Communication, New York: Springer-Verlag, 1997.

Алфавитный указатель

J

M

O

P

R

S

T

U

V

W

Y

Z

А

Б

В

Г

Д

Ж

З

И

К

Л

М

Н

О

П

P

C

Т

У

Ф

Ц

Ч

Э

Я

Эрик Брауде

Технология разработки программного обеспечения

Перевели с английского Е. Бочкарева, Д. Солнышков

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Руководитель проекта	*Е. Саргаева*
Научный редактор	*Ф. Новиков*
Литературный редактор	*Е. Бочкарева*
Художник	*Н. Биржаков*
Корректоры	*Н. Лукина, И. Смирнова*
Верстка	*Р. Гришанов*

Лицензия ИД № 05784 от 07.09.01.
Подписано в печать 22.12.03. Формат 70×100/16. Усл. п. л. 56,4375.
Тираж 4000 экз. Заказ № 1333.
ООО «Питер Принт». 196105, Санкт-Петербург, ул. Благодатная, д. 67в.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 953005 — литература учебная
Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

ЭРИК ДЖ. БРАУДЕ

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

КНИГИ, КОТОРЫЕ НЕ СТАРЕЮТ!

КЛАССИКА COMPUTER SCIENCE

Данное издание — не просто книга о программировании. Как о ней говорит сам автор, это «учебник по исследованию методов разработки программного обеспечения». В нем рассматриваются основные этапы процесса создания программного продукта, в том числе анализ предъявляемых к нему требований, составление технического задания, проектирование, собственно написание программы, включая интеграцию и тестирование программы в целом и ее сопровождение после установки у заказчика. Отдельные главы посвящены определению типа процесса разработки и управлению проектом, что особенно важно при работе в команде.

Как и в любой книге по программированию, особое внимание уделено примерам. Автор иллюстрирует принципы разработки программного обеспечения на примере создания ролевой видеоигры, используя широкие возможности для демонстрации образцов проектирования, параллельных процессов, состояний и нетривиальных графических пользовательских интерфейсов.

Книга предназначена для студентов и аспирантов, специализирующихся в области программного обеспечения, а также для профессиональных разработчиков, которые хотят расширить свои познания, усовершенствовать навыки и развить способности.

ISBN 5-94723-663-X

ПИТЕР®
WWW.PITER.COM
WILEY

ЭРИК
ДЖ. БРАУДЕ

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ