

Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

ФАКУЛЬТЕТ ДИСТАНЦИОННОГО ОБУЧЕНИЯ (ФДО)

М. М. Милихин, М. М. Рычагов

ПРОЕКТИРОВАНИЕ И АРХИТЕКТУРА ПРОГРАММНЫХ СРЕДСТВ

Учебное пособие

Томск
2015

УДК 004.41(075.8)
ББК 32.973.2-018я73
М 601

Рецензенты:

Жуковский О. И., канд. техн. наук, генеральный директор ООО «АОИ Софт»;
Гриценко Ю. Б., канд. техн. наук, доцент кафедры автоматизации обработки информации ТУСУРа.

Милихин М. М.

М 601 Проектирование и архитектура программных средств : учебное пособие / М. М. Милихин, М. М. Рычагов. — Томск : факультет дистанционного обучения ТУСУРа, 2015. — 138 с.

В учебном пособии подробно рассмотрены основные аспекты проектирования и разработки архитектуры программного обеспечения. Помимо вопросов создания проекта программной системы пособие содержит необходимые сведения по менеджменту качества программного обеспечения, выбору модели жизненного цикла системы и другим смежным вопросам, решение которых находит свое отражение в архитектурной документации.

УДК 004.41(075.8)
ББК 32.973.2-018я73

© Милихин М. М.,
Рычагов М. М., 2015
© Оформление.
ФДО, ТУСУР, 2015

ОГЛАВЛЕНИЕ

Введение	5
1 Программное средство	8
1.1 Понятия «Программное средство» и «Программное обеспечение» . .	8
1.2 Классификация программных средств	9
1.2.1 Системное программное обеспечение	11
1.2.2 Инструментарий технологии программирования	12
1.2.3 Прикладное программное обеспечение	12
2 Качество программного обеспечения	16
2.1 Международные стандарты качества	17
2.2 Отечественные стандарты качества программного обеспечения . . .	19
2.3 Модели качества программного обеспечения	19
3 Жизненный цикл программного обеспечения	24
3.1 Процессы жизненного цикла программного обеспечения	24
3.2 Модели жизненного цикла программного обеспечения	28
4 Формирование и анализ требований	38
4.1 Требования	38
4.2 Разработка и управление требованиями	45
4.2.1 Разработка требований	45
4.2.2 Разработка документации на требования к программному обеспечению	47
4.2.3 Управление требованиями	51
4.3 Как понять, что сбор требований завершен	52
5 Моделирование требований	54
5.1 Диаграмма прецедентов	55
5.2 Диаграмма состояний	65
5.3 Диаграмма деятельности	69
5.4 Диаграмма классов	74
5.5 Диаграммы последовательности	78
5.6 Диаграмма кооперации	82
5.7 Диаграмма компонентов	86
5.8 Диаграмма развертывания	90
6 Архитектура программного обеспечения	95
6.1 Понятие архитектуры программных средств	97
6.2 Целесообразность разработки и документирования архитектуры . . .	100

6.3	Проектирование архитектуры программных средств	105
6.4	Понятие архитектурного стиля	106
6.5	Комплексный подход к построению архитектуры	128
Заключение		130
Литература		131
Список условных обозначений и сокращений		135
Глоссарий		136

ВВЕДЕНИЕ

Проектированием программного обеспечения (системы) называется процесс определения архитектуры, компонентов, интерфейсов и других характеристик системы. Рассматриваемое как процесс, результатом которого является дизайн системы, проектирование есть инженерная деятельность в рамках жизненного цикла (в данном контексте — программного обеспечения), в которой надлежащим образом анализируются требования для создания описания внутренней структуры программного обеспечения, являющейся основой для конструирования программного обеспечения как такового [1]. Программный дизайн (как результат деятельности по проектированию) должен описывать архитектуру программного обеспечения (ПО), то есть представлять декомпозицию программной системы в виде организованной структуры компонент и интерфейсов между компонентами. Важнейшей характеристикой готовности дизайна является тот уровень детализации компонентов, который позволяет заняться их конструированием.

На текущем этапе развития компьютерных технологий проектирование играет особенно важную роль в процессах жизненного цикла (ЖЦ) создания программного обеспечения и, в свою очередь, зависит от используемой в рамках конкретного проекта модели ЖЦ ПО. Фактически, сегодня трудно представить возможность создания крупного программного продукта (ПП) без проведения предварительного детального анализа и тщательного изучения требований к ПП в их совокупности [2].

Целью настоящего учебного пособия является рассмотрение всех основных аспектов проектирования и разработки архитектуры программных средств для формирования целостного представления о процессе проектирования.

Проектирование программных систем можно рассматривать как деятельность, результат которой состоит из двух составных частей:

1. Архитектурный, или высокоуровневый, дизайн (*software architectural design, top-level design*) — описание высокоуровневой структуры и организации компонентов системы.
2. Детализированная архитектура (*software detailed design*), описывающая каждый компонент в том объеме, который необходим для конструирования.

Выходом этого процесса является набор моделей и артефактов, содержащих результаты решений, принятых по способам реализации требований в программном коде. В рамках настоящего пособия особое внимание уделено принципам формирования высокоуровневого дизайна. Разработка такого дизайна служит одновре-

менно нескольким целям, рассмотренным более подробно в главе 6, среди которых следует отдельно выделить задачи достижения установленных показателей надежности и качества ПО. Кроме того, так как проектирование опирается на определенные требования к конечному продукту, управление требованиями представляет собой основу процесса разработки любой архитектуры ПО.

Отметим также, что сегодня архитектура программных средств в большинстве случаев опирается на известные и хорошо проверенные архитектурные шаблоны, существующие для каждого класса ПО.

В настоящем учебном пособии рассмотрены все перечисленные аспекты проектирования и архитектуры. В первой главе приведены описание основных терминов из области программного обеспечения, используемых в пособии, и классификация программных средств. Вторая глава посвящена краткому рассмотрению современных стандартов качества в области программного обеспечения. В третьей главе описаны различные подходы к организации жизненного цикла программных средств, в зависимости от выбора которых изменяется и сам процесс проектирования. Четвертая глава содержит необходимые сведения для работы с требованиями. Пятая глава посвящена вопросам проектирования детализированной архитектуры (структуры) ПП. Наконец, в шестой главе дано определение высокоуровневой архитектуры ПО, а также освещены вопросы разработки и документирования архитектуры через призму архитектурных стилей.

Многие используемые в учебном пособии термины приведены одновременно на русском и на английском языках. Это сделано с целью упрощения поиска дополнительной информации в англоязычных источниках.

Соглашения, принятые в книге

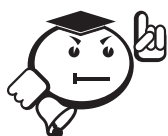
Для улучшения восприятия материала в данной книге используются пиктограммы и специальное выделение важной информации.



.....
Этот блок означает определение или новое понятие.



.....
 Этот блок означает внимание. Здесь выделена важная информация, требующая акцента на ней. Автор здесь может поделиться с читателем опытом, чтобы помочь избежать некоторых ошибок.



.....
 В блоке «На заметку» автор может указать дополнительные сведения или другой взгляд на изучаемый предмет, чтобы помочь читателю лучше понять основные идеи.



.....
Эта пиктограмма означает совет. В данном блоке можно указать более простые или иные способы выполнения определенной задачи. Совет может касаться практического применения только что изученного или содержать указания на то, как немного повысить эффективность и значительно упростить выполнение некоторых задач.
.....



..... Пример

.....
Эта пиктограмма означает пример. В данном блоке автор может привести практический пример для пояснения и разбора основных моментов, отраженных в теоретическом материале.
.....



.....
Контрольные вопросы по главе
.....

Глава 1

ПРОГРАММНОЕ СРЕДСТВО

1.1 Понятия «Программное средство» и «Программное обеспечение»

Начнем рассмотрение вопросов проектирования и архитектуры программных средств с уточнения термина «программное средство». В соответствии с ГОСТ 28806—90, пункт 2, раздел 1 [3] программное средство имеет следующее определение.



.....
Программное средство (ПС) (англ. *software*) — это объект, состоящий из программ, процедур, правил, а также, если предусмотрено, сопутствующих им документации и данных, относящихся к функционированию системы обработки информации.
.....

Программное средство представляет собой конкретную информацию, объективно существующую как совокупность всех значимых с точки зрения ее представлений свойств каждого из материальных объектов, содержащих в фиксированном виде эту информацию.

Понятие «Программное средство» включает в себя частный случай понятия «Программное обеспечение», которое в соответствии с пунктом 2 таблицы 1 из ГОСТ 19781—90 [4] имеет следующее определение.



.....
Программное обеспечение — это совокупность программ системы обработки информации и программных документов, необходимых для эксплуатации этих программ.
.....

Как видно из приведенных определений, программное средство является несколько более широким понятием, по сравнению с программным обеспечением, поскольку кроме набора программ включает в себя также порядок действий, применяемых для решения задач (процедуры, правила), однако в рамках данного учебного пособия мы будем рассматривать данные определения как синонимы и придерживаться термина «Программное обеспечение», как наиболее распространенного понятия, и его сокращенной аббревиатуры «ПО».

1.2 Классификация программных средств

Классификация программных средств позволяет выбрать в качестве основы спецификации требований, предъявляемых к разрабатываемому ПО, шаблон, содержащий базовые требования для любого программного обеспечения соответствующего класса (типа). Так, например, список требований к любому пакету офисных приложений имеет много общих элементов, и, в то же время, этот список будет сильно отличаться от набора требований, предъявляемых к операционным системам. Поэтому проектирование ПО логично начать с определения базовых требований, предъявляемых к программным продуктам соответствующего типа.

На сегодняшний день выделяют три основных класса программных средств [5, 6]:

- 1) системное ПО;
- 2) пакеты прикладных программ (прикладное ПО);
- 3) инструментарий технологии программирования (инструментальные средства для разработки ПО).



.....
Системное ПО (англ. *System Software*) — совокупность программ и программных комплексов для обеспечения работы компьютера и вычислительных сетей.
.....

Системное ПО направлено на создание операционной среды функционирования других программ, обеспечение надежной и эффективной работы самого компьютера и вычислительной сети, проведение диагностики и профилактики аппаратуры компьютера и вычислительных сетей, выполнение вспомогательных технологических процессов (копирование, архивация, восстановление файлов программ и БД и т. п.).



.....
Прикладное ПО — комплекс взаимосвязанных программ для решения задач определенного класса предметной области.
.....

Прикладное ПО служит программным инструментарием решения функциональных задач и является самым многочисленным классом ПО. В данный класс входят программные продукты, выполняющие обработку информации различных предметных областей. Рассматриваемые в главе 6 пособия архитектурные стили и шаблоны ориентированы на максимально эффективное применение именно в рамках прикладного ПО.



Инструментарий технологии программирования — это программные продукты, предназначенные для поддержки технологии программирования.

Инструментарий технологии программирования обеспечивает процесс разработки программ и включает специализированное ПО, которое является инструментальным средством разработки. ПО данного класса поддерживает все технологические этапы процесса проектирования, программирования, отладки и тестирования создаваемых программ. Пользователями данного ПО являются системные и прикладные программисты.

На рисунке 1.1 представлена общая классификация программного обеспечения [5]. Рассмотрим представленную классификацию более подробно.



Рис. 1.1 – Классификация программного обеспечения

1.2.1 Системное программное обеспечение

В составе системного ПО выделяют две основные составляющие:

- 1) базовое ПО;
- 2) сервисное ПО.

Базовое ПО — это минимальный набор программных средств, обеспечивающих работу компьютера. *Сервисное ПО* — это программы и программные комплексы, которые расширяют возможности базового ПО и организуют более удобную среду работы пользователя. Чаще всего сервисное ПО называют утилитами.

К базовому ПО относят операционные системы, операционные оболочки (текстовые, графические) и сетевые операционные системы.

Сервисное ПО по функциональному признаку можно разделить на следующие классы:

- программы диагностики работоспособности компьютера;
- антивирусные программы, обеспечивающие защиту компьютера, обнаружение и восстановление зараженных файлов;
- программы обслуживания дисков, обеспечивающие проверку качества поверхности магнитного диска, контроль сохранности файловой системы на логическом и физическом уровнях, сжатие дисков, резервное копирование данных на внешние носители и др.;
- программы архивирования данных, которые обеспечивают процесс сжатия информации с целью уменьшения объема памяти для ее хранения;
- программы обслуживания сети.



.....
Операционная система — это совокупность программных средств, обеспечивающих управление аппаратной частью компьютера и прикладными программами, а также их взаимодействием между собой и пользователем.
.....

Сетевая ОС — это комплекс программ, обеспечивающий обработку, передачу и хранение данных в сети. Сетевая ОС предоставляет пользователям различные виды сетевых служб (управление файлами, электронная почта, процессы управления сетью и т.д.), поддерживает работу в абонентских системах. Сетевые ОС используют архитектуру клиент-сервер или одноранговую архитектуру (эти термины будут рассмотрены подробно в главе 6).

Операционные оболочки представляют собой специальные программы, предназначенные для облегчения общения пользователя с командами ОС. Операционные оболочки имеют текстовый и графический варианты интерфейса конечного пользователя. Эти программы существенно упрощают задание управляющей информации для выполнения команд ОС, уменьшают сложность работы конечного пользователя [3].

Более подробно с классификацией и особенностями работы различных операционных систем можно ознакомиться в [6].

1.2.2 Инструментарий технологии программирования

Согласно классификации, представленной на рисунке 1.1, инструментарий технологии программирования включает в себя средства для создания приложений и средства для создания информационных систем.



.....
***Средства для создания приложений** — это совокупность языков и систем программирования, инструментальные среды пользователя, а также различные программные компоненты для отладки и поддержки создаваемых программ.*
.....

Инструментальная среда пользователя представляет собой специальные средства, встроенные в пакеты прикладных программ. К таким средствам относят библиотеки функций, процедур, объектов и методов обработки; макрокоманды; клавишные макросы; языковые макросы; конструкторы экранных форм и объектов; генераторы приложений; языки запросов высокого уровня; конструкторы меню.

Отдельного внимания в рамках рассмотрения вопросов проектирования ПО заслуживают *CASE-технологии* (*CASE — Computer-Aided System Engineering*). CASE-средства представляют собой программные комплексы, автоматизирующие весь технологический процесс анализа, проектирования, разработки и сопровождения сложных программных систем.

Средства CASE-технологий делятся на две группы:

- 1) встроенные в систему реализации — все решения по проектированию и реализации привязки к выбранной СУБД;
- 2) независимые от системы реализации — все решения по проектированию ориентированы на унификацию (приведение к единообразию) начальных этапов жизненного цикла программы и средств их документирования, обеспечивают большую гибкость в выборе средств реализации.

Использование CASE-технологий и специализированных программных средств позволяет автоматизировать процесс проектирования и разработки архитектуры, предоставляет возможность оптимизации процессов взаимодействия членов команды разработчиков ПО между собой, а также содержит инструменты создания проектной документации. Более подробно эти вопросы рассмотрены в главе 6 настоящего пособия.

Отметим, что в некоторых CASE-системах поддерживается кодогенерация программ — создание каркаса программ или создание полного продукта. Примерами программных продуктов для создания приложений являются Visual C++, Delphi, Visual Basic и т. д. [3].

1.2.3 Прикладное программное обеспечение

В рамках прикладного ПО выделяют проблемно-ориентированное ПО, ПО автоматизированного проектирования, методо-ориентированное ПО, офисное ПО и ПО общего назначения [3].

Проблемно-ориентированное ПО. Для некоторых предметных областей возможна типизация функций управления, структуры данных и алгоритмов обработки. Это вызвало разработку значительного количества программных систем одинакового функционального назначения: ПО автоматизированного бухгалтерского учета, финансовой деятельности, управления персоналом, управления производством, банковские информационные системы и т. д.

ПО автоматизированного проектирования предназначено для поддержки работы конструкторов и технологов, связанных с разработкой чертежей, схем, графическим моделированием и конструированием. Отличительными особенностями этого класса ПО являются высокие требования к аппаратному обеспечению, наличие библиотек встроенных функций, объектов, интерфейсов с графическими системами и БД (в качестве примера можно привести AutoCAD).

Методо-ориентированное ПО. Данный класс охватывает программные продукты, обеспечивающие независимо от предметной области и функций информационных систем математические, статистические и другие методы решения задач. Наиболее распространены методы математического программирования, решения дифференциальных уравнений, имитационного моделирования, исследования операций (Storm, SYSTAT, SAS и другие) [3].

Офисное ПО. Данный класс программных средств охватывает задачи управления деятельностью офиса:

- Органайзеры (планировщики) — ПО для планирования рабочего времени, составления протоколов встреч, расписаний, ведения записей и телефонной книжки. Примерами такого ПО являются калькулятор, записная книжка, часы, календарь.
- Программы-переводчики, средства проверки орфографии, распознавания текста.
- Коммуникационные пакеты, предназначенные для организации взаимодействия пользователей с удаленными абонентами или информационными ресурсами сети.
- Браузеры, средства создания WWW-страниц.
- Средства электронной почты.

В рамках *ПО общего назначения* можно выделить семь основных типов программных средств.

1. Системы управления базами данных (СУБД), обеспечивающие организацию и хранение локальных баз данных (БД) на автономно работающих компьютерах либо централизованное хранение БД на файл-сервере и сетевой доступ к ним. В современных СУБД содержатся элементы CASE-средств, в частности визуализирована схема БД, осуществлена автоматическая поддержка целостности БД при различных видах обработки (включение, удаление, модификация), предоставляются так называемые мастера, обеспечивающие поддержку процесса проектирования, а также созданы шаблоны (прототипы) структур БД, отчетов и форм.

2. Серверы БД — это ПО, предназначенное для создания и использования при работе в сети интегрированных БД в архитектуре клиент-сервер. Многопользовательские СУБД в сетевом варианте обработки информации хранят данные на файл-сервере, специально выделенном компьютере, но сама обработка ведется на рабо-

чих станциях. Серверы БД, в отличие от этого, большую часть обработки данных (хранение, поиск, извлечение и передачу данных клиенту) выполняют самостоятельно, одновременно обеспечивая данными большое число пользователей сети. Общим для различных видов серверов БД является использование реляционного языка SQL (*Structured Query Language*) для реализации запросов к данным. Большинство серверов БД поддерживает несколько платформ, широкий спектр протоколов передачи данных. К серверам БД предъявляются повышенные требования в вопросах обеспечения целостности данных, тиражирования данных по узлам сети и синхронного обновления.

3. Генераторы отчетов (серверы отчетов), обеспечивающие реализацию запросов и формирование отчетов в печатном или экранном виде в условиях сети с архитектурой клиент-сервер. Сервер отчетов подключается к серверу БД, использующему драйверы сервиса БД (Crystal Reports, Profit for Windows).

4. Текстовые процессоры, предназначенные для работы с текстовыми документами (Microsoft Word, Libre Office Writer). Развитием данного направления являются издательские системы.

5. Табличные процессоры, являющиеся удобной средой для вычислений конечным пользователем, содержат средства деловой графики, средства специализированной обработки (Microsoft Excel, Libre Office Calc).

6. Средства презентационной графики — специализированные программы, предназначенные для создания изображений и их показа на экране, подготовки слайд-фильмов, мультфильмов и их проектирования (Microsoft PowerPoint, Libre Office Impress).

7. Интегрированные пакеты — набор нескольких программных продуктов, функционально дополняющих друг друга, поддерживающих единые информационные технологии, реализованные на единой операционной и вычислительной платформе (Microsoft Office, Libre Office). Компоненты интегрированных пакетов могут работать изолированно друг от друга, но имеют общий интерфейс, благодаря этому упрощается их освоение конечными пользователями.

Программные средства мультимедиа. Основное значение данных программных средств — создание и использование аудио- и видеоинформации для расширения информационного пространства пользователя (различные БД компьютерных произведений искусства, библиотеки звуковых записей и т. д.).

Системы искусственного интеллекта включают в себя такие программные продукты, как:

- программы-оболочки для создания экспертных систем путем наполнения баз знаний и правил логического вывода;
- готовые экспертные системы для принятия решений в рамках определенных предметных областей;
- системы анализа и распознавания речи, текста и т. п. Примеры систем искусственного интеллекта: FIDE, MYSIN, Guru и др.

Настольные издательские системы. Данный класс ПО включает программы, обеспечивающие информационную технологию компьютерной издательской деятельности (например, PageMaker, CorelDraw, Adobe PhotoShop) [3]. Основными функциональными возможностями издательских систем являются форматирование

и редактирование текстов, автоматическая разбивка текста на страницы, компьютерная верстка печатной страницы, монтирование графики, подготовка иллюстраций и т. п.

В заключение отметим, что каждая из рассмотренных групп программных средств имеет общие требования, свойственные целому классу ПО, однако в то же время отдельно взятый программный продукт, как правило, обладает некоторыми функциональными возможностями или атрибутами качества, делающими его уникальным в своем классе. В следующей главе мы более подробно рассмотрим вопросы качества программного обеспечения, достижению которого в значительной степени посвящён весь процесс проектирования ПС, и вплотную подойдем к рассмотрению задачи проектирования функциональной составляющей программного обеспечения.



Контрольные вопросы по главе 1

1. Что такое программное обеспечение и программное средство? Чем отличаются эти понятия?
2. Перечислите три основных класса программных средств.
3. Какое программное обеспечение называется прикладным?
4. На какие классы можно разделить прикладное программное обеспечение по его функциональности?
5. Что называется инструментарием технологии программирования? Что такое CASE-средства и CASE-технологии?
6. В чем разница между базовым и сервисным ПО в рамках системного программного обеспечения?

Глава 2

КАЧЕСТВО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В современном мире, в связи с бурным развитием компьютерных технологий и обилием разрабатываемого программного обеспечения, вопрос обеспечения качества данного программного обеспечения становится все более и более актуальным. В настоящее время существует несколько определений качества в целом и качества программного обеспечения в частности. Мы рассмотрим наиболее широко применяемые в IT-индустрии.

Определение качества по стандарту ISO 9000 [7].

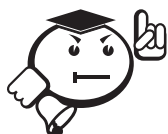


.....
***Качество** — это степень соответствия совокупности присущих характеристик требованиям.*
.....

Определение качества по ISO 9126 [8].



.....
***Качество** — это весь объем признаков и характеристик продукции или услуги, который относится к их способности удовлетворять установленным или предполагаемым требованиям.*
.....



.....
Один из наиболее известных в мире экспертов в области качества Фил Кросби в 1979 году дал определение качеству как «соответствию пользовательским требованиям» [9].
.....

Как видно из приведенных определений, под качеством принято понимать некое соответствие программного продукта выдвигаемым ему требованиям. О том,

что это за требования и как именно они «выдвигаются», мы поговорим в следующих главах, поскольку формирование и анализ требований являются ключевыми этапами проектирования ПО.

Качество программного обеспечения есть величина, отражающая, в каком объеме в программный продукт включен набор желаемых функций для повышения эффективности ПП в течение его жизненного цикла.



.....
 Для любой системы, использующей программное обеспечение, на этапе ее проектирования должны быть разработаны три вида спецификаций, такие как функциональные требования, требования к качеству и требования к ресурсам.

Качество включает в себя все характеристики и существенные особенности продукта или его функционирования, которые относятся к удовлетворению заданных спецификациями требований [10].

Рассмотрим применяемые в настоящее время стандарты качества.

2.1 Международные стандарты качества

Текущее положение дел, на момент написания данного пособия, в области стандартов качества ПО выглядит следующим образом. В 1986 году международной организацией по стандартизации ИСО (*International Organization for Standardization*, ISO) был принят стандарт ISO 8402 «Управление качеством и обеспечение качества», как основополагающий стандарт системы управления качеством на предприятии в целом. Данный стандарт не рассматривал программное обеспечение как уникальный вид продукции, поэтому на обеспечение качества программного обеспечения возлагались такие же требования, как и на другие материальные продукты.

Стандарт ISO 8402:1986 был пересмотрен и переиздан в 1994 году как стандарт ISO 8402:1994 с небольшими корректировками. Затем, в 2000 году, данный стандарт был и вовсе упразднен и заменен семейством стандартов ISO 9000, которые были разработаны с целью оказания помощи организациям всех видов и размеров при внедрении и обеспечении функционирования эффективных систем менеджмента качества. Наиболее значимыми стандартами серии ISO 9000 являются следующие [11]:

- ISO 9000. Стандарт описывает основные положения систем менеджмента качества и устанавливает терминологию для систем менеджмента качества.
- ISO 9001. Стандарт устанавливает требования к системам менеджмента качества для тех случаев, когда организация должна продемонстрировать возможность изготавливать продукцию, отвечающую требованиям потребителей и установленным к ней обязательным требованиям, и направлен на повышение удовлетворенности потребителей.
- ISO 9004. Стандарт содержит рекомендации по повышению результативности и эффективности системы менеджмента качества и предназначен для

улучшения деятельности организации и повышения удовлетворенности потребителей и других заинтересованных сторон.

ISO 9000 был переиздан как стандарт ISO 9000:2005 в 2005 году и пересмотрен в 2009 году. На момент написания данного пособия ведется разработка стандарта ISO/DIS 9000 «Системы менеджмента качества. Основные положения и словарь», который заменит стандарт ISO 9000:2005, однако пока этого не произошло, стандарт ISO 9000:2005 «Системы менеджмента качества» остается актуальным.

Вышеупомянутое семейство стандартов, как и стандарт ISO 8402, затрагивает лишь общие положения системы менеджмента качества на предприятии, поэтому в 1991 году был выпущен стандарт ISO/IEC 9126 «Информационные технологии. Оценка программных продуктов. Характеристики качества и руководящие положения по их применению», который регламентировал систему менеджмента качества программного обеспечения.

В 2001 году стандарт ISO/IEC 9126 также был пересмотрен и выделен в семейство стандартов, включающее в себя такие, как [12]:

- ISO/IEC 9126—1 «Программирование. Качество продукта. Часть 1. Модель качества».
- ISO/IEC 9126—2 «Программирование. Качество продукта. Часть 2. Внешние метрики».
- ISO/IEC 9126—3 «Программирование. Качество продукта. Часть 3. Внутренние метрики».
- ISO/IEC 9126—4 «Программирование. Качество продукта. Часть 4. Качество при использовании показателей».

Кроме семейства стандартов ISO/IEC 9126 существует семейство стандартов оценки программного продукта ISO/IEC 14598 [12]:

- ISO/IEC 14598—1:1999 «Информационные технологии. Оценка программного продукта. Часть 1. Общий обзор».
- ISO/IEC 14598—2:2000 «Информационные технологии. Оценка программного продукта. Часть 2. Планирование и руководство».
- ISO/IEC 14598—3:2000 «Информационные технологии. Оценка программного продукта. Часть 3. Процесс для разработчиков».
- ISO/IEC 14598—4:1999 «Информационные технологии. Оценка программного продукта. Часть 4. Процесс для заказчика».
- ISO/IEC 14598—5:1998 «Информационные технологии. Оценка программного продукта. Часть 5. Процесс для блоков».
- ISO/IEC 14598—6:2001 «Информационные технологии. Оценка программного продукта. Часть 6. Документирование модулей оценки».

Однако процесс разработки стандартов обеспечения качества программного обеспечения на этом не останавливается. Международная организация по стандартизации ИСО в настоящее время ведет разработку стандартов требований к качеству систем и программного обеспечения и их оценку (*Systems and software Quality Requirements and Evaluation* — SQuaRE), которые объединят ISO/IEC 9126 и ISO/IEC 14598 в семействе стандартов ISO/IEC 250mm. На момент написания

данного пособия семейство стандартов ISO/IEC 250mm все еще находится в разработке, поэтому мы рекомендуем дождаться их официальной публикации перед внедрением в повседневную практику.

2.2 Отечественные стандарты качества программного обеспечения

В Российской Федерации приняты свои нормативные документы, регламентирующие оценку качества программных средств. Среди них следует отметить следующие:

- ГОСТ 28806—90 «Качество программных средств. Термины и определения».
- ГОСТ 28195—89 «Оценка качества программных средств. Общие положения».
- ГОСТ ISO 9000—2011 «Системы менеджмента качества. Основные положения и словарь».
- ГОСТ Р ИСО/МЭК 9126—93 «Информационная технология. Оценка программной продукции. Характеристики качества и руководства по их применению».

Как видно из названий, в России существуют как собственные нормативные документы, так и переведенные международные, однако различия между ними не столь велики. ГОСТ Р ИСО/МЭК 9126 является для Российской Федерации актуальным стандартом оценки качества программной продукции и фактически является переводом стандарта ISO/IEC 9126 [8].

2.3 Модели качества программного обеспечения

Кроме стандартов, регламентирующих понятия и метрики качества ПО, существует множество моделей оценки качества программного обеспечения. Кратко рассмотрим основные из них.

Модель МакКола (1977 г.). Является первой широко распространенной моделью качества и была предназначена для определения качества программного продукта через его различные характеристики. В модели выделяется одиннадцать факторов качества, которые можно разделить на три направления — операции с продуктом, его использование (корректность, надежность, эффективность, целостность, практичность), ревизия продукта (тестируемость, гибкость, сопровождаемость) и переносимость продукта (мобильность, многократное использование, функциональная совместимость).

Кроме факторов качества, существуют 23 критерия качества (например, независимость от аппаратной платформы, унификация данных, функциональная полнота и подобные), которые и определяют наличие того или иного фактора качества в программном обеспечении. Более подробно с моделью МакКола можно ознакомиться в [13–15].

Модель Бозма (1978 г.). Данная модель похожа на модель МакКола, но факторы качества тут несколько другие и группируются по назначению:

- мобильность;
- практичность (надежность, эффективность, эргономичность проектирования);
- сопровождаемость (тестируемость, понятность, модифицируемость).

Как и в модели МакКола, в данной модели описан набор критериев качества, определяющих наличие факторов в программном продукте. Подробное описание модели представлено в [16].

Модель FURPS/FURPS+. Название модели представляет собой акроним, обозначающий категории требований к качеству ПО: *functionality* (функциональность), *usability* (практичность), *reliability* (надежность), *performance* (производительность), *supportability* (эксплуатационная пригодность).

Символ «+» добавляет к модели такие составляющие, как ограничения проекта, интерфейс, требования к выполнению, физические требования, требования к лицензированию.

Основной концепцией, лежащей в основе модели качества FURPS, является декомпозиция характеристик программного обеспечения на две категории требований: *функциональные* (F) и *нефункциональные* (URPS) требования. Эти выделенные категории могут быть использованы как в качестве требований к программному продукту, так и в оценке качества ПП. В настоящее время модель FURPS+ широко используется в разработке программного обеспечения и при идентификации требований к разрабатываемой системе целесообразно использование FURPS+ модели в качестве универсального контрольного перечня характеристик ПО [17].

Модель качества ISO/IEC 9126.



.....
Прежде чем перейти к рассмотрению данной модели, следует напомнить, что стандарты ISO/IEC 9126 будут заменены стандартами ISO/IEC 250mm, однако на момент написания данного пособия актуальными являются стандарты ISO/IEC 9126.
.....

Эта модель не является прямым расширением ранее предложенных. В ней оценка качества ПО основана на трехуровневом представлении:

- 1) цели (*goals*) — то, что мы хотим видеть в ПО;
- 2) атрибуты (*attributes*) — свойства ПО, показывающие приближение к целям;
- 3) метрики (*metrics*) — количественные характеристики степени наличия атрибутов.

В рамках модели выделено 6 целей: функциональность (*functionality*), надежность (*reliability*), практичность или удобство использования (*usability*), эффективность (*efficiency*), сопровождаемость (*maintainability*), переносимость или мобильность (*portability*). Цели подразделяются на 21 атрибут качества.

В 2001 году этот стандарт был пересмотрен и расширен. В него было добавлено 6 дополнительных атрибутов качества: привлекательность как атрибут прак-

тичности и степень соответствия стандартам для каждой из целей, кроме функциональности.

Ниже приведен полный список атрибутов качества ПО по стандарту ISO 9126:

1) функциональность:

- пригодность к определенной работе (*suitability*);
- точность, правильность (*accuracy*);
- способность к взаимодействию (*interoperability*);
- соответствие стандартам и правилам (*compliance*);
- защищенность (*security*);

2) надежность:

- зрелость, завершенность (обратна к частоте отказов) (*maturity*);
- устойчивость к отказам (*fault tolerance*);
- способность к восстановлению работоспособности при отказах (*recoverability*);
- соответствие стандартам надежности (*reliability compliance*, добавлен в 2001 г.);

3) практичность, удобство использования:

- понятность (*understandability*);
- удобство обучения (*learnability*);
- работоспособность (*operability*);
- привлекательность (*attractiveness*, добавлен в 2001 г.);
- соответствие стандартам практичности (*usability compliance*, добавлен в 2001 г.);

4) эффективность:

- временные характеристики (*time behaviour*);
- использование ресурсов (*resource utilisation*);
- соответствие стандартам эффективности (*efficiency compliance*, добавлен в 2001 г.);

5) сопровождаемость:

- анализируемость (*analyzability*);
- изменяемость, удобство внесения изменений (*changeability*);
- риск возникновения неожиданных эффектов при внесении изменений (*stability*);
- контролируемость, удобство проверки (*testability*);
- соответствие стандартам сопровождаемости (*maintainability compliance*, добавлен в 2001 г.);

б) переносимость, мобильность:

- адаптируемость (*adaptability*);
- устанавливаемость, удобство установки (*installability*);
- способность к сосуществованию с другим ПО (*coexistence*);
- удобство замены другого ПО данным (*replaceability*);
- соответствие стандартам переносимости (*portability compliance*, добавлен в 2001 г.).

Принятые в 2001 году части 2 и 3 стандарта ISO 9126 определяют набор метрик качества ПО. В качестве примера таких метрик приведем следующие.

1. Полнота реализации функций. Используется для измерения пригодности.
2. Корректность реализации функций. Используется для измерения пригодности.
3. Отношение числа обнаруженных дефектов к прогнозируемому. Используется для определения зрелости.
4. Отношение числа проведенных тестов к общему их числу. Используется для определения зрелости.
5. Отношение числа доступных проектных документов к указанному в их списке. Используется для измерения анализируемости.

Заметим, что третья и четвертая метрики из приведенных показывают, что качество ПО зависит не только от самого ПО как объекта материального мира, но и от его восприятия заинтересованными лицами: разработчиками, пользователями, заказчиками и др. Действительно, не изменяя самого ПО, можно повысить его качество только за счет проведения некоторых еще не проведенных тестов, поскольку значение четвертой метрики при этом увеличится.

Показатели качества играют огромную роль в проектировании ПО, позволяя разработчикам сконцентрировать внимание и ресурсы на наиболее важных проблемах. При этом для каждого приложения необходимо использовать собственный набор показателей. Например, если вопросы безопасности и производительности необходимо учесть при разработке практически каждого приложения, то проблемы возможности взаимодействия или масштабируемости стоят далеко не перед всеми проектами. Первым делом необходимо понять поставленные требования и сценарии развертывания, чтобы знать, какие показатели качества важны для создаваемого приложения (требования к ПО будут подробно рассмотрены в главе 4).

На сегодняшний день не существует универсального стандарта или методики оценки программного обеспечения, как нет и обязательных требований проведения оценки — это сугубо выбор разработчика. Однако не стоит забывать, что определенные с заказчиком критерии качества и верно выбранная методика могут повысить конкурентоспособность программного обеспечения и сэкономить огромные денежные средства при сопровождении данного ПО.

Вопросы влияния архитектуры на качество конечного программного продукта рассмотрены в главе 6 настоящего пособия.



Контрольные вопросы по главе 2

1. Что такое качество? Какие международные стандарты качества в области разработки программного обеспечения Вы знаете?
2. Какое семейство международных стандартов качества в области разработки ПО является актуальным на сегодняшний день? Назовите основные стандарты, входящие в это семейство.
3. Какой из отечественных стандартов качества разработки ПО является актуальным?
4. Какие модели качества ПО Вам известны?
5. Какая концепция качества ПО лежит в основе модели FURPS?
6. Перечислите основные элементы трехуровневого представления модели качества ISO/IEC 9126.

Глава 3

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В авторитетном словаре программной инженерии IEEE Std 610.1290 «*IEEE Standart Glossary of Software Engineering Terminology*» [18] записано: «Жизненный цикл программного обеспечения определяется как период времени, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации».

Проектирование программного обеспечения является неотъемлемой и важной частью его ЖЦ, а подход к решению задач разработки архитектуры ПО во многом определяется выбранной моделью жизненного цикла.

В данной главе будут рассмотрены основные характеристики используемых в настоящее время моделей жизненного цикла, приведены их преимущества, недостатки и сценарии использования.

3.1 Процессы жизненного цикла программного обеспечения

Актуальную классификацию процессов программной инженерии задают международный стандарт *ISO/IEC 12207:2008 «System and software engineering — Software life cycle processes»* и его российский аналог ГОСТ Р ИСО/МЭК 12207—2010 [19]. Данные стандарты, используя устоявшуюся терминологию, устанавливают общую структуру процессов жизненного цикла программных средств, на которую можно ориентироваться в программной индустрии. Стандарты определяют процессы, виды деятельности и задачи, которые используются при приобретении программного продукта или услуги, а также при поставке, разработке, применении по назначению, сопровождении и прекращении применения программных продуктов.

В соответствии с данными стандартами все процессы жизненного цикла программного обеспечения разделяются на семь групп. Все процессы в данных стандартах описываются в терминах цели и желаемых выходов, списков действий и задач, которые необходимо выполнять для достижения желаемых результатов. В данном пособии мы рассмотрим каждый процесс очень кратко. Более подробную информацию можно найти в [19].

Процессы в контексте системы

Процессы соглашения определяют действия, необходимые для выработки соглашений между двумя организациями. Если реализуется процесс приобретения, то он обеспечивает средства для проведения деловой деятельности с поставщиком продуктов, предоставляемых для применения в функционирующей системе, услугах поддержки этой системы или элементах системы, разработанных в рамках проекта. Если реализуется процесс поставки, то он обеспечивает средства для проведения проекта, в котором результатом является продукт или услуга, предоставляемые приобретающей стороне.

Процессы соглашения включают в себя процессы приобретения и процесс поставки.

Процессы организационного обеспечения проекта осуществляют менеджмент возможностей организаций приобретать и поставлять продукты или услуги через инициализацию, поддержку и управление проектами. Эти процессы обеспечивают ресурсы и инфраструктуру, необходимые для поддержки проектов, и гарантируют удовлетворение организационных целей и установленных соглашений. Они не претендуют на роль полной совокупности деловых процессов, реализующих менеджмент деловой деятельности организации.

Процессы организационного обеспечения проекта включают в себя:

- 1) процесс менеджмента модели жизненного цикла;
- 2) процесс менеджмента инфраструктуры;
- 3) процесс менеджмента портфеля проектов;
- 4) процесс менеджмента людских ресурсов;
- 5) процесс менеджмента качества.

В стандарте [19] проект выбран как основа для описания процессов, относящихся к планированию, оценке и управлению. Принципы, связанные с этими процессами, могут применяться в любой области менеджмента организаций.

Существуют две категории процессов проекта. Процессы менеджмента проекта используются для планирования, выполнения, оценки и управления продвижением проекта. Процессы поддержки проекта обеспечивают выполнение специализированных целей менеджмента. Обе категории процессов проекта описаны ниже.

Процессы менеджмента проекта применяются для создания и развития планов проекта, оценки фактического выполнения и продвижения относительно плановых заданий и управления выполнением проекта вплоть до полного его завершения. Отдельные процессы менеджмента проекта могут привлекаться в любое время жизненного цикла и на любом уровне иерархии проекта в соответствии с планами проекта или возникновением непредвиденных событий. Процессы менеджмента

проекта применяются на уровне строгости и формализации, зависящих от риска и сложности проекта, и включают в себя процесс планирования проекта и процесс управления и оценки проекта.

Процессы поддержки проекта формируют специфическую совокупность задач, ориентированных на выполнение специальных целей менеджмента. Все эти процессы очевидны при осуществлении менеджмента любой иницилируемой деятельности, располагаясь по нисходящей от организации в целом вплоть до отдельного процесса жизненного цикла и его задач:

- 1) процесс менеджмента решений;
- 2) процесс менеджмента рисков;
- 3) процесс менеджмента конфигурации;
- 4) процесс менеджмента информации;
- 5) процесс измерений.

Технические процессы используются для определения требований к системе, преобразования требований в полезный продукт, для разрешения постоянного копирования продукта (где это необходимо), применения продукта, обеспечения требуемых услуг, поддержания обеспечения этих услуг и изъятия продукта из обращения, если он не используется при оказании услуги.

Технические процессы определяют деятельность, которая дает возможность реализовывать организационные и проектные функции для оптимизации пользы и снижения рисков, являющихся следствием технических решений и действий. Эта деятельность обеспечивает возможность продуктам и услугам обладать такими свойствами, как своевременность и доступность, результативность затрат, а также функциональность, безотказность, ремонтпригодность, продуктивность, приспособленность к применению и другими качественными характеристиками, требуемыми приобретающими и поддерживающими организациями. Она также предоставляет возможность продуктам и услугам соответствовать ожиданиям или требованиям гражданского законодательства, включая факторы здоровья, безопасности, защищенности и факторы, относящиеся к окружающей среде.

Технические процессы состоят из следующих процессов.

1. Определение требований правообладателей.
2. Анализ системных требований.
3. Проектирование архитектуры системы.
4. Процесс реализации.
5. Процесс комплексирования системы.
6. Процесс квалификационного тестирования системы.
7. Процесс инсталляции программных средств.
8. Процесс поддержки приемки программных средств.
9. Процесс функционирования программных средств.
10. Процесс сопровождения программных средств.
11. Процесс изъятия из обращения программных средств.

Специальные процессы программных средств

Процессы реализации программных средств используются для создания конкретного элемента системы (составной части), выполненного в виде программного средства. Эти процессы преобразуют заданные характеристики поведения, интерфейсы и ограничения на реализацию в действия, результатом которых становится системный элемент, удовлетворяющий требованиям, вытекающим из системных требований.

Процесс реализации программных средств включает в себя несколько специальных процессов более низкого уровня.

1. Процесс анализа требований к программным средствам.
2. Процесс проектирования архитектуры программных средств.
3. Процесс детального проектирования программных средств.
4. Процесс конструирования программных средств.
5. Процесс комплексирования программных средств.
6. Процесс квалификационного тестирования программных средств.

Процессы поддержки программных средств предусматривают специально сфокусированную совокупность действий, направленных на выполнение специализированного программного процесса. Любой поддерживающий процесс помогает процессу реализации программных средств как единое целое с обособленной целью, внося вклад в успех и качество программного проекта. Существует восемь таких процессов.

1. Процесс менеджмента документации программных средств.
2. Процесс менеджмента конфигурации программных средств.
3. Процесс обеспечения гарантии качества программных средств.
4. Процесс верификации программных средств.
5. Процесс валидации программных средств.
6. Процесс ревизии программных средств.
7. Процесс аудита программных средств.
8. Процесс решения проблем в программных средствах.

Группа *процессов повторного применения программных средств* состоит из трех процессов, которые поддерживают возможности организации использовать повторно составные части программных средств за границами проекта. Эти процессы уникальны, поскольку, в соответствии с их природой, они используются вне границ какого-либо конкретного проекта.

Процессами повторного применения программных средств являются:

- 1) процесс проектирования доменов;
- 2) процесс менеджмента повторного применения активов;
- 3) процесс менеджмента повторного применения программ.

Все процессы жизненного цикла согласно ГОСТ Р ИСО/МЭК 12207—2010 представлены на рисунке 3.1.

В зависимости от особенностей организации описанных процессов можно выделить пять основных моделей жизненного цикла программного обеспечения.

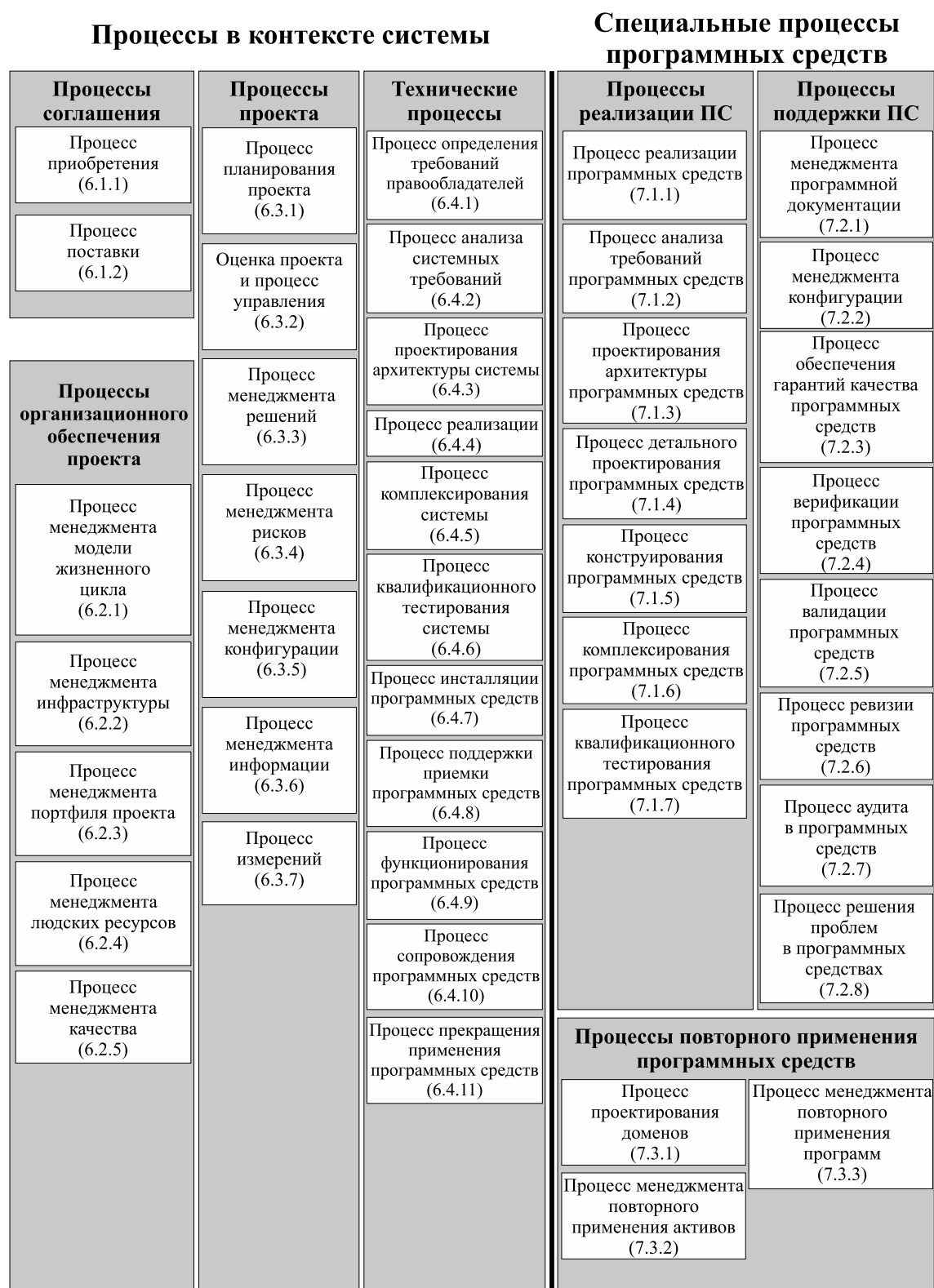


Рис. 3.1 – Группы процессов жизненного цикла

3.2 Модели жизненного цикла программного обеспечения

Процесс жизни любой системы или программного продукта может быть описан посредством модели жизненного цикла, состоящей из стадий. Модели могут

использоваться для представления всего жизненного цикла от замысла до прекращения применения или для представления части жизненного цикла, соответствующей текущему проекту. Модель жизненного цикла представляется в виде последовательности стадий, которые могут перекрываться и (или) повторяться циклически в соответствии с областью применения, размером, сложностью, потребностью в изменениях и возможностях. Каждая стадия описывается формулировкой цели и выходов. Процессы и действия жизненного цикла отбираются и исполняются на этих стадиях для полного удовлетворения поставленных целей. Различные организации могут использовать разные стадии в пределах жизненного цикла. Однако каждая стадия реализуется организацией, ответственной за эту стадию, с надлежащим рассмотрением информации, имеющейся в планах жизненного цикла и решениях, принятых на предшествующих стадиях. Аналогичным образом организация, ответственная за текущую стадию, ведет записи принятых решений и записи допущений, относящихся к последующим стадиям данного жизненного цикла.

В соответствии с ISO/IEC 12207:2010 понятие «Модель жизненного цикла» имеет следующее определение [19].



.....
***Модель жизненного цикла** — это структура процессов и действий, связанных с жизненным циклом, организуемых в стадии, которые также служат в качестве общей ссылки для установления связей и взаимопонимания сторон.*
.....

Рассмотрим наиболее известные модели жизненного цикла программного обеспечения и приведем их основные преимущества и недостатки.

Каскадная (водопадная) модель

Данная модель предполагает строго последовательное (во времени) и однократное выполнение всех фаз проекта с жестким (детальным) предварительным планированием в контексте predetermined или однажды и целиком определенных требований к программной системе.

Согласно каскадной модели существует семь основных этапов ЖЦ.

1. Определение требований.
2. Проектирование.
3. Конструирование.
4. Воплощение (реализация).
5. Тестирование и отладка.
6. Инсталляция.
7. Поддержка.

Обычно некоторые этапы объединяются или опускаются, и чаще всего модель представляется в виде, в котором она представлена на рисунке 3.2.

Переход к следующей фазе осуществлялся только в тот момент, когда вся работа на предыдущем этапе была завершена. Каждый этап завершается выпуском

полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков. Однако такой подход применим лишь для небольших систем, требования которых можно однозначно сформулировать еще до начала разработки. Для реальных современных систем этого сделать невозможно, поскольку такие системы просто не укладываются в такую жесткую схему. Кроме того, каскадной модели свойственна постоянно возникающая потребность в возврате к предыдущим фазам с целью корректировки или пересмотра уже принятых решений. Таким образом, реальная каскадная модель принимает вид, представленный на рисунке 3.3.

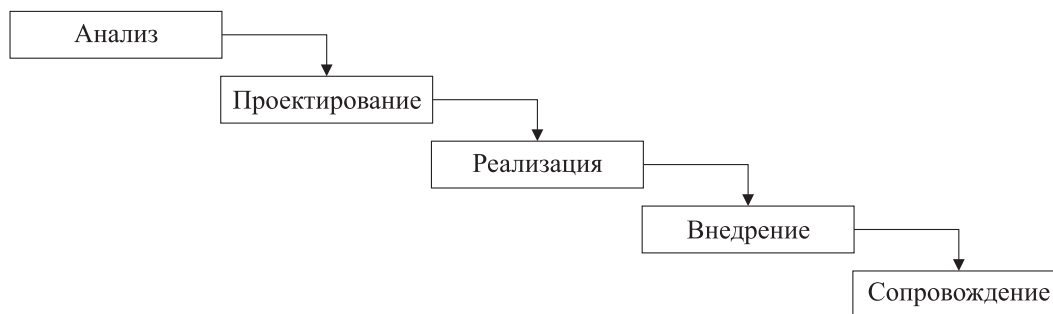


Рис. 3.2 – Каскадная модель жизненного цикла

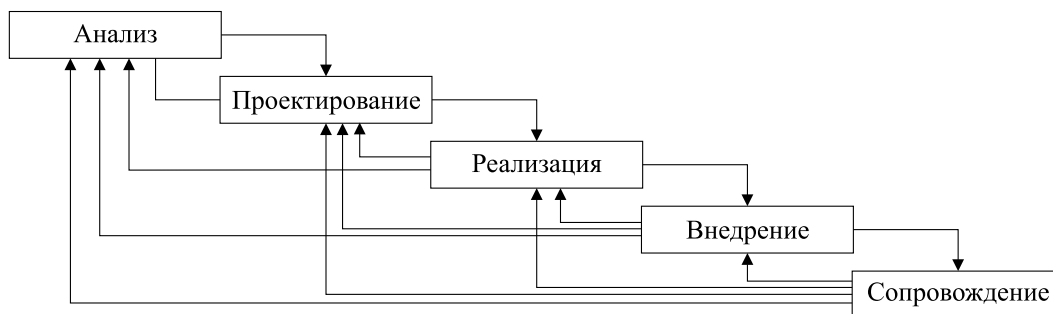


Рис. 3.3 – Реальная каскадная модель жизненного цикла

Преимущества каскадной модели.

1. Проста для понимания и использования.
2. Проста в управлении благодаря жесткости модели — каждая фаза имеет ожидаемый результат.
3. Фазы выполняются строго последовательно и каждая в один момент времени. Фазы не накладываются.
4. Хорошо применима к маленьким проектам, требования к которым могут быть однозначно сформулированы.

Недостатки каскадной модели.

1. Если проект находится на этапе тестирования, очень сложно вернуться обратно и внести изменения на предыдущих этапах.
2. Заказчик не получает работающего программного обеспечения до тех пор, пока не закончится его полная реализация.

3. Водопадной модели свойственны большие риски и неопределенности.
4. Модель слабо подходит для программных комплексов или объектно-ориентированных проектов, а также продолжительных во времени проектов.
5. Модель практически не применима к проектам, в которых имеется большой риск изменения требований.

Каскадную модель можно использовать при выполнении следующих условий:

- требования хорошо известны, понятны и фиксированы;
- имеется стабильное описание продукта;
- технологии понятны;
- требуемые ресурсы свободно доступны;
- проект достаточно непродолжителен во времени.

Сегодня водопадная модель находит наиболее широкое применение в условиях, когда ведущая роль отводится качеству документации и строгой поэтапной реализации проекта, например в рамках проектов, финансируемых из средств федерального бюджета.

Инкрементная модель

В рамках инкрементной модели имеет место множество циклов разработки, которые превращают модель в подобие повторяющихся каскадных моделей. Все циклы разделяются на маленькие, легко управляемые модули. Каждый из модулей проходит фазы выявления требований, проектирования, разработки и тестирования. Рабочая версия продукта получается уже после выполнения первого модуля, то есть гораздо раньше, чем в случае водопадной модели. Каждый новый модуль добавляет к программному продукту новые функциональные возможности. Процесс продолжается, пока все запланированные модули не будут выполнены и система не будет закончена (рис. 3.4).

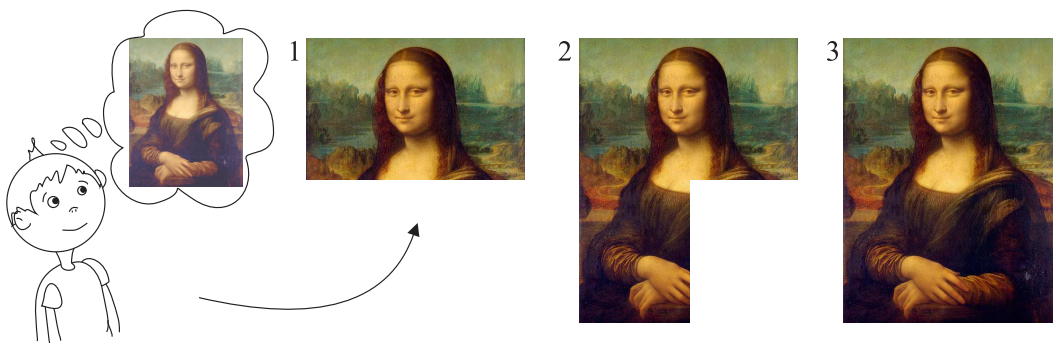


Рис. 3.4 – Пример выполнения работы в рамках инкрементной модели

При использовании инкрементной модели возможно уже на ранней стадии предоставить рабочую версию программного обеспечения, которое выполняет поставленные функции и прошло стадию тестирования. Дальнейший функционал наращивается постепенно, такими же законченными кусочками.

Диаграмма инкрементной модели представлена на рисунке 3.5. После выявления требований разработка делится на модули (сборки), и данные модули реализуются отдельно друг за другом, давая каждый законченную часть программного обеспечения.

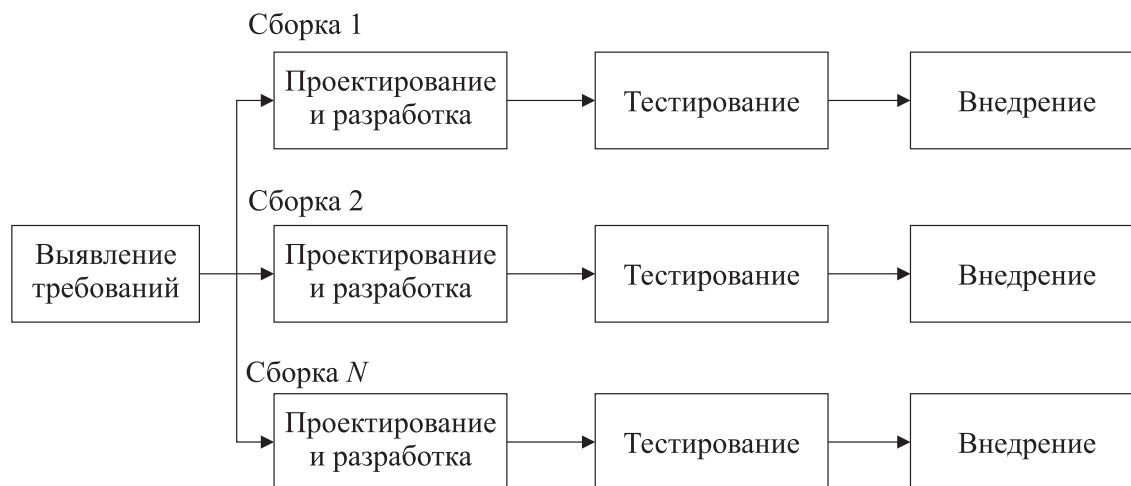


Рис. 3.5 – Инкрементная модель жизненного цикла

Преимущества инкрементной модели.

1. Рабочий программный продукт получается достаточно рано.
2. Модель более гибкая по сравнению с каскадной, благодаря чему требует меньше затрат при изменении масштабов продукта и требований.
3. Тестирование продукта упрощается благодаря малым итерациям.
4. Снижается стоимость первоначальной доставки продукта.
5. Облегчаются задачи управления рисками, поскольку они разбиваются на отдельные итерации.

Недостатки инкрементной модели.

1. Необходимость детального планирования перед началом разработки.
2. Необходимость ясного и полного понимания целой системы до того, как она разделится на части.
3. Общая стоимость применения модели выше, чем у водопадной (под стоимостью понимаются не только денежные ресурсы, требуемые на разработку в рамках модели, но также временные и людские ресурсы).

Модель можно использовать в следующих случаях:

- 1) требования к системе строго определены и понятны;
- 2) общие требования определены, однако некоторые детали могут появиться со временем;
- 3) существует необходимость получить минимальный продукт как можно быстрее;
- 4) в рамках проекта используются новые непроверенные технологии;
- 5) проект характеризуется большим количеством рискованных целей и особенностей.

Итерационная модель

Итерационная модель жизненного цикла позволяет не начинать разработку с полностью определенных требований. Вместо этого, разработка может начинаться даже с минимальным набором требований, который будет наращиваться в процессе разработки. Выявление требований повторяется до тех пор, пока продукт не будет готов (рис. 3.6).

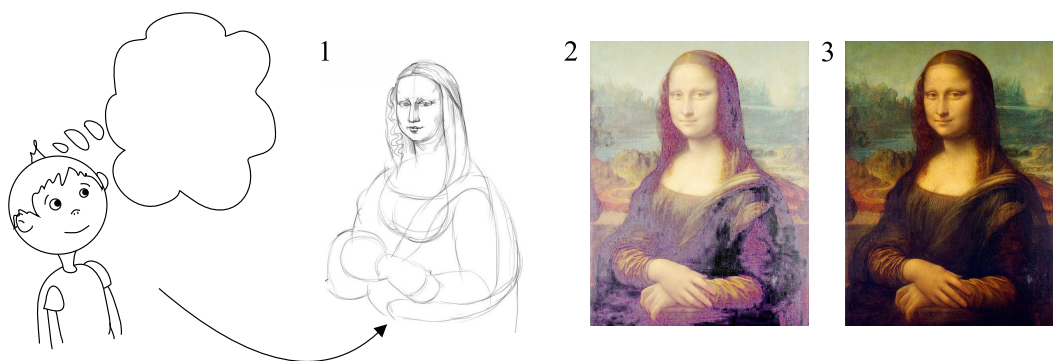


Рис. 3.6 – Пример выполнения работы в рамках итерационной модели

Основное отличие итерационной модели от инкрементной заключается в том, что в случае инкрементной модели наращивание функциональности системы происходит завершенными кусками, в то время как в итеративной модели постепенно улучшается весь продукт в целом.

Диаграмма итерационной модели представлена на рисунке 3.7. После разработки предшествующей версии продукта выявляются новые требования и осуществляется переход к новому циклу разработки, пока продукт не будет готов.

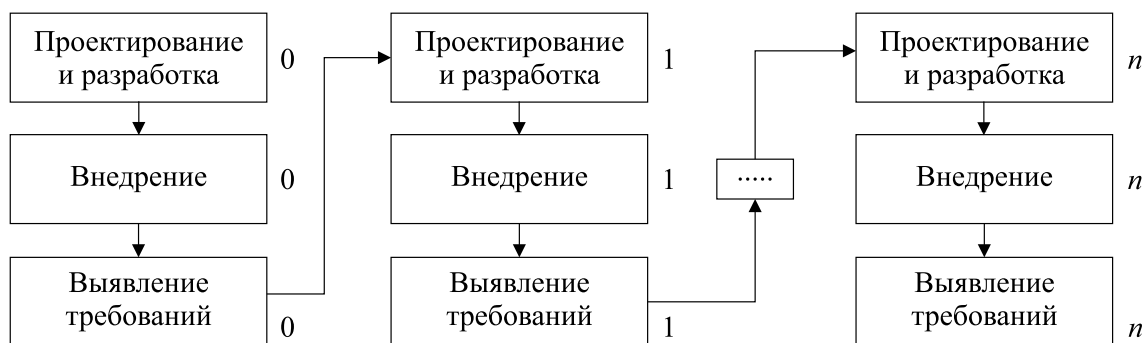


Рис. 3.7 – Итерационная модель жизненного цикла

Преимущества итерационной модели.

1. В рамках итерационной модели возможно создать макет приложения до того, как будет завершено проектирование всего продукта. Создается каркас ПО, после чего конечный продукт улучшается на основании этого каркаса.
2. Проблемы и риски могут быть выявлены на ранних стадиях.
3. Итерационная модель обеспечивает надежную обратную связь с пользователем.

Недостатки итерационной модели обуславливаются тем, что фазы в итерации не накладываются друг на друга, и в случае, если требования собраны не в полном объеме для всего ЖЦ, могут возникнуть проблемы при построении архитектуры или проектировании.

Итерационную модель можно использовать, если требования к системе строго определены и понятны. Она подходит для достаточно крупных проектов, при этом основные требования должны быть определены до начала разработки системы, однако некоторые детали вполне могут добавляться со временем.

Спиральная модель

Спиральная модель также похожа на инкрементную модель, но больший уклон в ней сделан на анализ рисков. Данная модель жизненного цикла включает четыре фазы: планирование, анализ рисков, разработку и оценку. Разработка в каждой своей итерации (в данной модели итерации называются спиралями) проходит через эти фазы. Считается, что в основании спирали, которое начинается в фазе планирования, все требования собраны и риски оценены. Каждая последующая спираль основывается на предыдущей спирали.



При использовании спиральной модели стоит помнить о том, что с каждой спиралью увеличиваются затраты на проект.

В течение данной *фазы планирования* происходит сбор требований. В *фазе анализа рисков* осуществляется попытка определить риски и найти альтернативные решения. Результатом этой фазы является прототип продукта. Если в течение данной фазы выявляются какие-либо риски, то предлагается и разрабатывается альтернативное решение.

Фаза разработки предназначена непосредственно для разработки продукта и его тестирования. Последней фазой служит *фаза оценки*, в рамках которой клиент оценивает получившийся ПП с целью выявления новых требований. Фаза оценки завершается датой, когда продукт вступает в новую спираль.

Схематично суть спиральной модели представлена на рисунке 3.8.

Спиральная модель ЖЦ хорошо подходит для больших и критически важных проектов. Большой анализ рисков способствует меньшему их возникновению в дальнейшем. Строгий контроль согласованности с клиентом и документирования обеспечивает успешное достижение поставленных требований. В проект на любом этапе может быть добавлена новая функциональность, а рабочий программный продукт получается достаточно рано.

Недостатки спиральной модели.

1. Использование модели может оказаться весьма дорогостоящим.
2. Анализ рисков требует больших усилий.
3. Успех проекта сильно зависит от фазы анализа рисков.
4. Модель плохо применима для маленьких проектов.

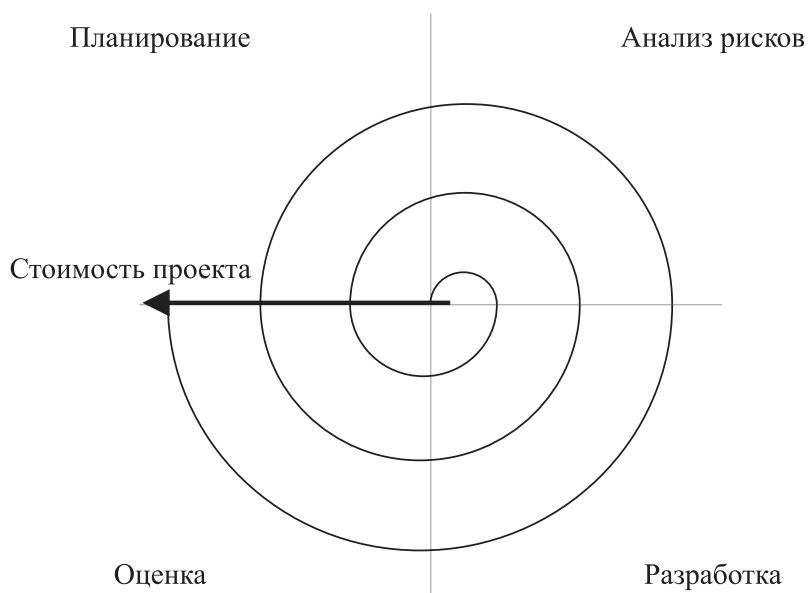


Рис. 3.8 – Диаграмма спиральной модели

Спиральную модель следует использовать, если оценка стоимости и рисков очень важна. Она хорошо подходит для средних по масштабу проектов и проектов с высокими рисками, а также в случаях, когда пользователи не уверены в том, что им нужно. Данная модель оказывается эффективной в случае выпуска совершенно нового продукта или если в процессе разработки ожидаются значительные изменения в требованиях к конечному продукту.

Гибкая модель (Agile)

Гибкая модель разработки является разновидностью инкрементной модели. Программное обеспечение также разрабатывается быстрыми циклами. Результатами работы являются маленькие релизы, основанные каждый на предыдущем. Для обеспечения качества программного обеспечения каждый релиз проходит фазу тестирования. Основное отличие от инкрементной модели заключается в том, что завершённый цикл разработки представлен не просто минимальным объемом кода, а полноценным релизом с законченной функциональностью, который можно поставлять пользователям. Модель находит широкое применение в проектах, для которых время разработки имеет важнейшее значение. *Extreme Programming (XP)* и *SCRUM* являются одними из самых популярных методологий разработки программного обеспечения, использующих данную модель жизненного цикла. Диаграмма гибкой модели представлена на рисунке 3.9.

При использовании гибкой модели ЖЦ клиент, как правило, удовлетворен постоянной и быстрой поставкой релизов программного обеспечения. Людям и взаимодействию в команде придается большее значение, чем самому процессу разработки и инструментам, — клиенты, разработчики и тестировщики постоянно взаимодействуют друг с другом. Непрерывное внимание уделяется техническому совершенствованию и проектированию, происходит постоянная адаптация к изменяющимся обстоятельствам, и даже запоздалые изменения требований не наносят существенного вреда проекту.

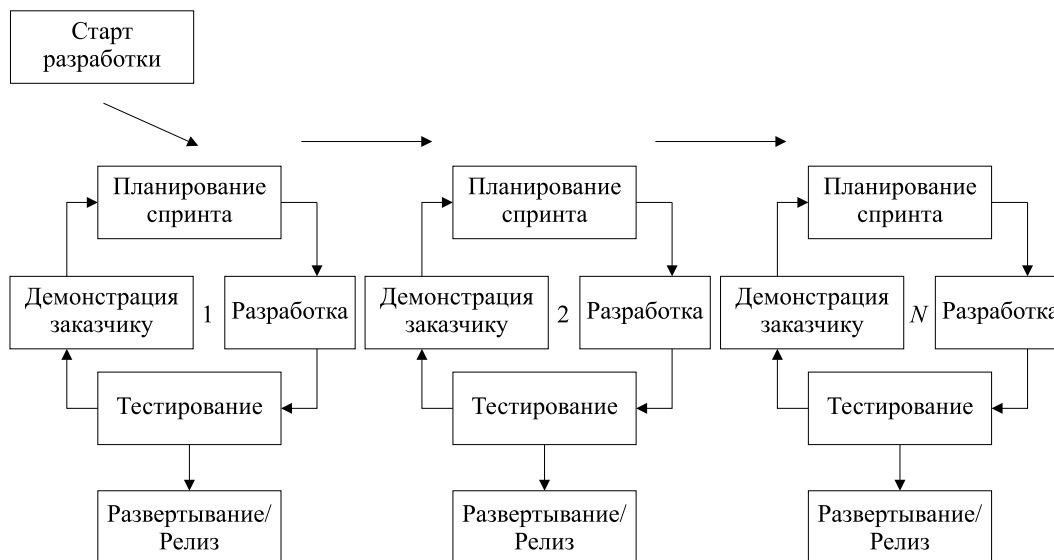


Рис. 3.9 – Диаграмма гибкой модели

К недостаткам гибкой модели можно отнести тот факт, что в случае некоторых программных продуктов, особенно больших проектов, трудно оценить первоначальное достаточное усилие в самом начале разработки. Проектированию и документированию при использовании *agile*-технологий уделяется меньше внимания, чем в классических моделях ЖЦ.

Отдельно следует отметить, что проект может легко выйти за пределы временных рамок, если клиент до конца не понимает, что он хочет получить в итоге. Кроме того, только квалифицированные разработчики способны принимать решения, необходимые в процессе разработки. Поэтому использование гибкой модели в команде новичков при отсутствии опытного разработчика обычно не дает высоких результатов.

Когда использовать данную модель.

1. При необходимости включать в проект новые изменения. Они могут быть включены без особых затрат благодаря часто повторяющимся шагам разработки.
2. Чтобы включить новую особенность в процесс, разработчики потеряют всего лишь день или даже пару часов, чтобы откатиться назад и включить эту особенность.
3. В отличие от каскадной модели, в гибкой модели не требуется глубокого планирования, чтобы начать разработку. Данная модель предполагает динамичное изменение потребностей пользователей. Изменения обсуждаются, и функциональные возможности могут быть добавлены или удалены на основе этих обсуждений.
4. Разработчики и другие заинтересованные стороны получают больше свободы в своих действиях, чем если бы использовалась более жесткая последовательность разработки программного обеспечения. Это позволяет им откладывать некоторые решения на более поздний срок для более эффективной реализации таких решений (например, при ожидании новой версии определенного компонента).

В заключение отметим, что выбор используемой модели жизненного цикла во многом определяется характеристикой выявленных требований к конечному программному продукту. В главах 4 и 5 мы более подробно рассмотрим процесс управления требованиями в рамках проектирования программного обеспечения.



Контрольные вопросы по главе 3

1. Перечислите основные процессы жизненного цикла ПО. Что такое технические процессы жизненного цикла ПО?
2. Назовите известные Вам модели жизненного цикла ПО.
3. Перечислите основные характеристики каскадной модели ЖЦ ПО.
4. В чем отличие между инкрементной и итерационной моделями ЖЦ?
5. Каковы основные преимущества спиральной модели ЖЦ ПО?
6. Когда выгодно использовать гибкую модель ЖЦ?

Глава 4

ФОРМИРОВАНИЕ И АНАЛИЗ ТРЕБОВАНИЙ

4.1 Требования

Задачи управления требованиями являются основополагающими в рамках проектирования ПО и разработки его архитектуры. Выявленные во время проведения анализа предметной области требования определяют как список функциональных возможностей приложения, так и показатели качества, характеризующие конечный программный продукт. Фактически, проектирование любой программной системы начинается именно с определения и анализа требований различных заинтересованных сторон.

Определим границы понятия требования. Иан Коммервилл и Пит Соьер в 1997 году сформулировали следующее определение требований [22].



.....
***Требования** — это спецификация того, что должно быть реализовано. В них описано поведение системы, свойства системы или ее атрибуты. Они могут служить ограничениями в процессе разработки системы.*
.....

Это определение учитывает самые разные типы информации, которые в совокупности называются требованиями. Требования охватывают как видение пользователя, так и внешнее поведение системы, а также представление разработчика о некоторых внутренних характеристиках. Они включают как поведение системы в определенных условиях, так и свойства, которые делают систему полезной и даже доставляющей удовольствие конечным операторам [23].

Термин «требования» является обширным понятием и включает в себя бизнес-требования, бизнес-правила, ограничения, внешние требования к интерфейсу, ха-

рактические, функциональные требования, нефункциональные требования, атрибуты качества, системные требования, пользовательские требования.

Рассмотрим некоторые типы требований более подробно [23].

Бизнес-требования (*business requirements*) описывают, почему организации нужна такая программная система, то есть цели, которые организация намерена достичь с ее помощью. Основное их содержание — бизнес-цели организации или клиента, заказывающих систему. Как правило, бизнес-требования высказывают те, кто финансируют проект, покупатели системы, управляющий реальными пользователями, отдел маркетинга или ответственный за концепцию продукта. Обычно бизнес-требования записываются в форме документа о концепции и границах (*vision and scope document*). К другим руководящим документам, которые еще иногда используют в этом качестве, относят устав проекта (*project charter*), вариант использования (*business case*) или документ рыночных требований (*market requirements document*).



Пример

Пример бизнес-требования. Авиакомпания хочет на 25% снизить затраты на сотрудников у стойки в аэропорту. В процессе достижения этой цели может возникнуть идея поставить терминал, который пассажиры будут использовать для самостоятельной регистрации в аэропорту.

Пользовательские требования (*user requirements*) описывают цели или задачи, которые пользователи должны иметь возможность выполнять с помощью продукта, который, в свою очередь, должен приносить пользу кому-то. Область пользовательских требований также включает описания атрибутов или характеристик продукта, которые важны для удовлетворения пользователей. К отличным способам представления этого вида требований относятся варианты использования, пользовательские истории и таблицы «событие — отклик». В идеале эту информацию предоставляют реальные представители пользователей. Пользовательские требования описывают, что пользователь должен иметь возможность делать с системой. Важно помнить, что в большинстве проектов есть несколько классов пользователей, а также других участников, потребности которых тоже надо выявить.



Пример

Примером сценария использования является регистрация на рейс с использованием веб-сайта или терминала в аэропорту. Будучи сформулированным в виде пользовательской истории, то же пользовательское требование может звучать так: «Как пассажир я хочу зарегистрироваться на рейс, чтобы можно было сесть на самолет».

Функциональные требования (*functional requirements*) определяют, каким должно быть поведение продукта в тех или иных условиях. Они определяют, что разработчики должны создать, чтобы пользователи смогли выполнить свои задачи (пользовательские требования) в рамках бизнес-требований. Функциональные требования описываются в форме традиционных утверждений со словами «должен» или «должна». Бизнес-аналитик документирует функциональные требования в спецификации требований к ПО (*Software Requirements Specification, SRS*), где описывается так полно, как необходимо, ожидаемое поведение системы. Спецификация требований к ПО используется при разработке, тестировании, гарантии качества продукта, управлении проектом и в связанных с проектом функциях. Этот артефакт называют по-разному: документ бизнес-требований, функциональная спецификация, документ требований и т.п. Спецификация требований к ПО может представлять собой отчет, сгенерированный на основе информации, хранимой в средстве управления требованиями.



Пример

У пассажира должна быть возможность распечатать посадочные талоны на все рейсы, на которые он зарегистрировался, или если в профиле пассажира не указаны предпочтения по выбору места, система резервирования должна сама назначить ему место.

Термин **«системные требования»** (*system requirements*) описывает требования к продукту, который содержит многие компоненты или подсистемы. Говоря о системе, мы подразумеваем программное обеспечение или подсистемы ПО и оборудования. Люди и процессы тоже являются частью системы, поэтому определенные системные функции могут распространяться и на людей. Иногда термин «системные требования» используется для обозначения подробных требований к программной системе. Хороший пример «системы» — рабочее место кассира в супермаркете. В нем есть сканер штрих-кода, интегрированный с весами, а также ручной сканер штрих-кода. У кассира есть клавиатура, монитор и выдвижной ящик-касса. Есть также устройство чтения кредитных карточек и клавиатура для ввода ПИН-кода и чтения карт постоянного покупателя. На рабочем месте может быть до трех принтеров: для печати чека, квитанции кредитной карты и купонов на скидку. Все эти устройства взаимодействуют под управлением программного обеспечения. На основе требований к системе или продукту в целом бизнес-аналитик формулирует конкретную функциональность, которую должны поддерживать тот или иной компонент или подсистема, а также интерфейсы взаимодействия между ними.

Бизнес-правила (*business rules*) включают корпоративные политики, правительственные постановления, отраслевые стандарты и вычислительные алгоритмы. Бизнес-правила не являются сами требованиями к ПО, потому что они находятся за пределами любой системы ПО. Однако они часто налагают ограничения, определяя, какими функциями должна обладать система, подчиняющаяся

соответствующим правилам. Иногда, как в случае с корпоративными политиками безопасности, бизнес-правила становятся источником атрибутов качества, которые реализуются в функциональности. Следовательно, Вы можете отследить происхождение конкретных функциональных требований вплоть до соответствующих им бизнес-правил.

Согласно *Business Rules Group* [24] определение бизнес-правила звучит следующим образом.



.....

С точки зрения бизнеса **«бизнес-правило»** — это указание на существование обязательств относительно поведения, действия, принятого порядка или процедуры в определенной деятельности или отрасли»; с точки зрения информационной системы **«бизнес-правило»** — это указание, определяющее или ограничивающее определенный аспект бизнеса. Оно предназначается для установления бизнес-структуры или для управления и влияния на бизнес-деятельность».

.....

Для организации бизнес-правил предлагается множество разных схем классификации. Простейшая из них, представленная на рисунке 4.1, состоит из пяти типов бизнес-правил и пригодна для использования в большинстве случаев. Отдельно можно выделить шестую категорию бизнес-правил — термины — важные для бизнеса слова, фразы и аббревиатуры. Термины можно группировать по фактическим бизнес-правилам. Их также удобно хранить в словаре.



Рис. 4.1 – Классификация бизнес-правил



.....

Факты — это верные утверждения о бизнесе на определенный момент времени.

.....

Факты описывают связи и отношения между важными бизнес-терминами. Факты о сущностях данных, важных для системы, иногда применяют в моделях данных.

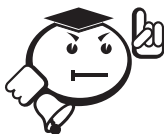


Пример

Примеры фактов:

- на каждый химический контейнер нанесен уникальный штрих-код;
- оплачивается доставка каждого заказа;

- со стоимости доставки налог с продаж не берется;
- если лицо, приобретающее химикат по невозмещаемому авиабилету, изменяет маршрут доставки, ему потребуется уплатить дополнительный сбор;
- книги, размер которых превышает 16 дюймов, размещаются в разделе широкоформатных книг библиотеки.



В компании есть бесчисленное количество фактов. Если собирать не относящиеся к делу факты, бизнес-анализ быстро застопорится. Даже если эти факты правильные, не всегда очевидно, как команда разработчиков должна использовать эту информацию. Сосредоточьтесь на фактах, находящихся в границах проекта, и не старайтесь собрать весь массив бизнес-знаний. Старайтесь связать каждый факт с входными и выходными данными контекстной диаграммы, событиями системы, известными объектами данных или конкретными пользовательскими требованиями.



Ограничение определяет, какие операции не может выполнять система и ее пользователи.

Среди слов и фраз, которые часто применяются при описании ограничивающего бизнес-правила, отметим такие, как «должен», «не должен», «не может» и «только определенные люди или роли могут выполнять определенные действия».



Пример

Приведем несколько примеров ограничений.

- Постоянный посетитель библиотеки может отложить для себя до 10 книг.
- Экипажи коммерческих авиарейсов должны каждые 24 часа отдыхать не менее 8 часов.
- Веб-приложения не должны содержать никаких HTML-тегов или атрибутов, не соответствующих стандарту HTML 5.



Правило, при определенных условиях инициирующее выполнение определенных действий, называется **активатором операции** (action enabler).

Человек может выполнять такие действия вручную, или правило может управлять некоторыми программными функциями, благодаря которым приложение при выполнении определенных условий реализует нужную модель поведения. Условия, определяющие выполнение операции, иногда представляют собой сложную комбинацию значений «истина» и «ложь», выполняющихся для нескольких отдельных условий. Выражение вида «Если <некоторое условие верно или наступило определенное событие>, то <что-то произойдет>» — это ключ, который описывает активатор операции.



Пример

Приведем примеры активаторов операций.

- Если в наличии имеются контейнеры с запрошенным химикатом, их следует предложить запросившему химикат лицу.
- В последний день квартала должны генерироваться отчеты для Управления по безопасности труда (OSHA) и Агентства по защите окружающей среды (EPA) о хранении и использовании химикатов в этом квартале.
- Если срок хранения контейнера с химикатом истек, об этом необходимо уведомить лицо, у которого в данный момент находится контейнер.

Вывод (inference), иногда его еще называют предположительным знанием или производным фактом, создает новый факт на основе других фактов. Выводы зачастую записывают в формате «если-то», применяемом также при записи бизнес-правил, активирующих операции; тем не менее раздел «то» вывода включает в себе факт, а не действие.



Пример

Приведем несколько примеров выводов.

- Если платеж не поступил в течение 30 календарных дней с момента отправки счета, то счет считается просроченным.
- Если поставщик не может поставить заказанный товар в течение пяти дней с момента получения заказа, то заказ считается невыполненным.
- Химикаты с токсичностью агента LD50 ниже 5 мг на килограмм массы мыши считаются опасными.

Пятый класс бизнес-правил определяют *вычисления (computations)*, преобразующие существующие данные в новую информацию с использованием математических формул и алгоритмов. Многие вычисления выполняются по внешним для предприятия правилам, например по формулам удержания подоходного налога.



Пример

Несколько примеров вычислительных бизнес-правил, выраженных в текстовой форме.

- Плата за доставку по суши в пределах страны для заказов весом свыше 2 фунтов составляет 4,75 доллара плюс 12 центов за полную или неполную унцию.
- Общая стоимость заказа вычисляется как сумма стоимости всех заказанных товаров, за вычетом скидок на количество, плюс государственные и местные налоги, действующие в округе, куда будет доставлен товар, плюс стоимость доставки и плюс необязательный страховой сбор.
- Цена единицы товара снижается на 10% при заказе от 6 до 10 единиц, на 20% — при заказе от 11 до 20 единиц и на 30% — при заказе свыше 20 единиц.

В дополнение к функциональным требованиям спецификация требований программного обеспечения должна включать в себя нефункциональные требования [23].

Атрибуты качества (quality attributes), называемые также параметрами качества или требованиями по уровню обслуживания, представляют собой описание различных измерений характеристик продукта, таких как производительность, доступность и переносимость, которые важны для пользователей или для разработчиков и тех, кто будет обслуживать систему. Другие нефункциональные требования описывают внешние интерфейсы между системой и внешним миром. Речь идет о подключениях к другим программным системам, аппаратным устройствам и пользователям, а также коммуникационных интерфейсах. Ограничения (*constraints*) проектирования и реализации накладывают границы на возможности выбора разработчика при проектировании продукта.



Характеристика (feature) — это набор логически связанных функциональных требований, которые представляют ценность для пользователя и удовлетворяют бизнес-цели.

Желательные характеристики продукта, которые перечисляет клиент, не эквивалентны тем, что входят в список необходимых для решения задач пользователей. В качестве примеров характеристик продуктов можно привести избранные страницы или закладки веб-браузера, средства проверки орфографии, запись макрокоманды, автоматическое обновление определений вирусов в антивирусной программе. Характеристики могут охватывать множество пользовательских требований, и для каждого варианта необходимо, чтобы множество функциональных требований было реализовано для выполнения пользователем его задач.

4.2 Разработка и управление требованиями

Путаница в терминологии возникает, как только речь заходит о требованиях, и затрагивает даже то, что именно называть этим словом. Некоторые авторы называют всю эту предметную область разработкой технических условий, другие употребляют термин «управление требованиями» (*requirements management*), а третьи считают эту деятельность подмножеством более общей предметной области бизнес-анализа [23].

Авторы пособия предлагают разделить область разработки технических условий (рис. 4.2) на разработку требований (*requirements development*) и управление требованиями (*requirements management*).

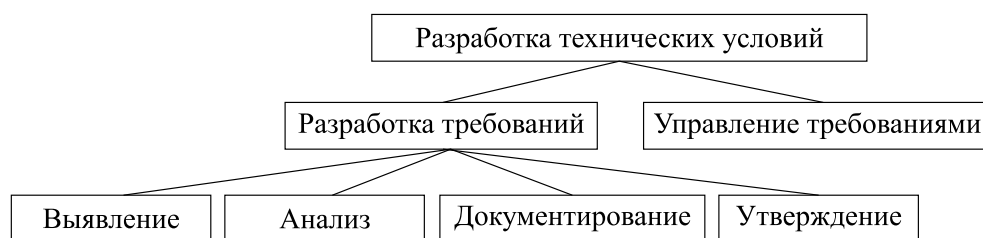


Рис. 4.2 – Составные части предмета разработки технических условий

4.2.1 Разработка требований

На практике действия по разработке требований редко удастся выполнить за один проход, необходимые действия, как правило, выполняются попеременно, поэтапно и часто повторяются (рис. 4.3).

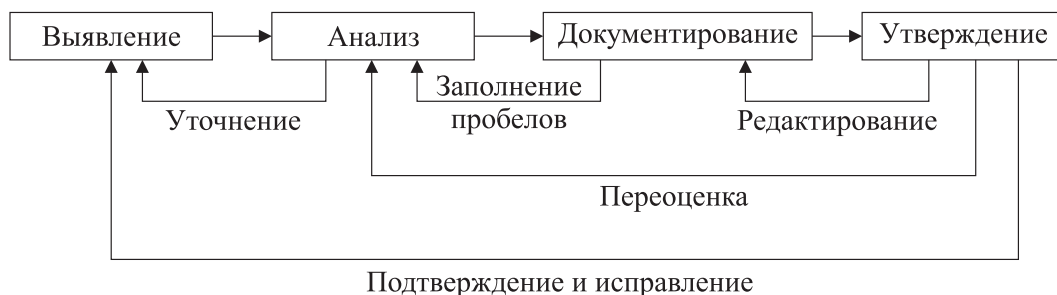


Рис. 4.3 – Итеративный процесс формулирования требований

Выявление требований

Все требования к программному обеспечению соответствуют одному из трех основных уровней: бизнес-уровень, пользовательский и функциональный уровни. Требования каждого уровня собираются из разных источников на различных этапах работы над проектом, имеют различные цели и аудиторию и должны документироваться по-разному. Также следует выявить и собрать нефункциональные

характеристики, например предполагаемое качество в разных измерениях, из соответствующих источников. Для повышения эффективности процесса сбора требований можно использовать следующие приемы [23]:

- определение концепции продукта и границ проекта;
- определение классов пользователей и их характеристик;
- выбор сторонника продукта (*product champion*) в каждом классе пользователей;
- проведение фокус-групп типичных пользователей;
- работа с пользователями для выяснения назначения продукта;
- определение системных событий и реакции на них;
- проведение интервью для выявления требований;
- проведение совместных семинаров;
- наблюдение за пользователями на рабочих местах;
- раздача опросных листов;
- анализ документов;
- изучение отчетов о проблемах работающих систем с целью поиска новых идей;
- повторное использование требований.

Анализ требований

Анализ требований подразумевает запись требований в форме, понятной всем заинтересованным лицам, а также тщательное исследование требований на предмет ошибок, пробелов и других недостатков. Кроме того, анализ включает разбиение высокоуровневых требований на более детальные, создание прототипов, анализ осуществимости и согласование приоритетов. Цель анализа — достаточно качественно и подробно описать требования, позволяющие менеджерам реалистично оценить все затраты на проект, а техническому персоналу — начать проектирование, разработку и тестирование.

Очень ценно представлять отдельные требования несколькими способами, например в текстовой и графической форме или одновременно в виде требований и тестов. Это позволит выявить их особенности и проблемы, незаметные при представлении одним способом. Также это помогает всем заинтересованным лицам выработать согласованное представление об итогах разработки продукта. Анализ требований включает в себя такие аспекты, как моделирование среды приложения, создание пользовательского интерфейса и технических прототипов, анализ осуществимости требований, определение приоритетов требований, создание словаря данных, моделирование требований, анализ интерфейсов между системой и внешним миром, распределение требований по подсистемам.

Документирование требований

Суть документирования требований заключается в документировании требований различных типов единообразным, доступным и поддающимся проверке способом так, чтобы они были понятными целевой аудитории [23]. Зафиксировать бизнес-требования можно в положении о концепции и границах. Пользовательские требования обычно представляют в виде вариантов применения или пользовательских историй. Подробные функциональные и нефункциональные требования к ПО фиксируются в спецификации требований к ПО (*Software Requirements Specification*, SRS) или другом хранилище, таком как средство управления требованиями. Процесс документирования требований включает в себя следующие этапы:

- внедрение шаблонов документов требований;
- определение источников требований;
- присвоение уникальных идентификаторов всем требованиям;
- документирование бизнес-правил;
- определение нефункциональных требований.

Документирование требований будет более подробно рассмотрено в главе 5.

Утверждение требований

Утверждение требований гарантирует, что все положения требований корректны, отражают желаемые качественные характеристики и удовлетворяют потребностям клиента [23]. Может оказаться, что требования, в документации к ПО выглядевшие превосходно, при реализации оказываются содержащими двусмысленности и неопределенности. Если требования должны стать надежной основой для проектирования и итоговой проверки системы посредством системного тестирования или приемочного тестирования пользователями, эти проблемы необходимо устранить. Процесс утверждения требований включает в себя рецензирование, тестирование, определение критериев приемки и моделирование требований.

4.2.2 Разработка документации на требования к программному обеспечению

При документировании требований к ПО определяются функции и возможности, которыми должно обладать ПО, а также все необходимые ограничения. Документация должна содержать достаточно подробное описание поведения системы при различных условиях, а также необходимые показатели качества системы, такие как производительность, безопасность и удобство использования. Документирование требований служит основой для дальнейшего планирования, дизайна и кодирования, а также базой для тестирования пользовательской документации. Однако разработанная документация, являясь основой для разработки архитектуры ПО, не должна содержать подробности дизайна, проектирования, тестирования и управления проектом за исключением известных ограничений дизайна и реализации.

Следует отметить, что не обязательно составлять документацию требований для всего продукта до начала его разработки, но, в то же время, требования для

каждой итерации должны быть зафиксированы до ее начала. Такой подход часто используется при применении итерационной, спиральной или гибкой модели ЖЦ ПО. Отклики об использовании ранних итераций позволяют скорректировать оставшуюся часть проекта в соответствии с новыми или уточненными требованиями. Однако при работе над любым проектом необходимо достичь базового соглашения по каждому набору требований до начала их реализации разработчиками. Выработка базового соглашения или базовой версии требований (*baselining*) представляет собой процесс преобразования реализуемых требований к ПО в то, что будет изучаться и одобряться. При работе с согласованным набором требований снижается вероятность недопонимания и ненужных переделок.

Структурируйте и составляйте документацию требований к ПО таким образом, чтобы все заинтересованные в проекте лица смогли в ней разобраться. Ниже приведены советы, как сделать требования ясными и понятными [23].

Требования к именованию. У каждого требования должен быть уникальный и неизменный идентификатор. Это позволит Вам ссылаться на определенные требования в запросе на изменения, в хронологии изменений, в перекрестных ссылках или матрице связей требований. При этом также упрощается повторное использование требований в нескольких проектах.

Нумерация по порядку. Самый простой способ нумерации — присвоить каждому требованию уникальный порядковый номер, например UR-9 или FR-26. Серийные средства управления требованиями присваивают такой идентификатор, когда пользователь добавляет новое требование в базу данных. Префикс обозначает тип требования, например FR означает «*functional requirement*» (то есть «функциональное требование»). При удалении требования номер повторно не используется, поэтому читателю не придется путаться между исходным требованием FR-26 или новым требованием FR-26.

Иерархическая нумерация. Это наиболее распространенный способ. Если функциональные требования приводятся в разделе 3.2 спецификации, то все их номера будут начинаться с 3.2. Чем больше цифр, тем больше уровень детализации требования, так что сразу понятно, что 3.2.4.3 является потомком по отношению к 3.2.4. Этот способ отличается простотой, компактностью и очевидностью.

Каждая организация, специализирующаяся на разработке ПО, обычно принимает один или несколько стандартных шаблонов документации требований к ПО для использования в проектах. С примерами таких шаблонов можно ознакомиться в [25, 26].



.....

Если вы беретесь за проекты различных типов и размеров, от конструирования новой объемной системы до небольших улучшений уже работающих систем, позаботьтесь для проектов каждого крупного класса завести отдельный шаблон документации.

.....

Определение приоритетов требований

Даже проект средних масштабов может иметь десятки пользовательских требований и сотни функциональных требований — слишком много, чтобы классифици-

ровать их аналитически и последовательно. Чтобы не потерять контроль над ситуацией, выберите для определения приоритетов определенный уровень абстракции — функции, варианты использования, пользовательские истории или функциональные требования. В рамках одного варианта использования некоторые альтернативные направления могут иметь больший приоритет, чем остальные. Вы имеете возможность сначала определить приоритеты на уровне функций, а затем назначить приоритеты функциональным требованиям отдельно для каждой функции. Это поможет вам отделить основную функциональность от усовершенствований, которые можно реализовать позже или вообще убрать. Не стоит забывать и о низкоприоритетных требованиях, хотя пока нет причин анализировать их более подробно. Их приоритет может впоследствии измениться, и если разработчики знают о них сейчас, им проще планировать будущую модификацию.

В процессе определения приоритетов могут применяться такие характеристики, как ценность для клиента, бизнес-ценность, технические или бизнес-риски, затраты, сложность реализации, время выпуска на рынок, соответствие нормативам или политикам, конкурентное преимущество и договорные обязательства. Алан Дейвис [27] указывает, что для успешного определения приоритетов требуется понимание шести характеристик:

- 1) потребности клиентов;
- 2) относительная важность требований для клиентов;
- 3) последовательность, в которой должны предоставляться функции;
- 4) требования, служащие основой для других требований, и другие связи между требованиями;
- 5) потребности реализовывать какие-то группы требований целиком;
- 6) затраты на удовлетворение каждого требования.

Приведем несколько приемов определения приоритетов [23].

Методы определения приоритетов

Включать или не включать. Простейший метод определения приоритетов заключается в том, что группа заинтересованных лиц просматривает список требований и по каждому принимает простое бинарное решение: включать требование в следующий выпуск или нет? При этом следует держать в уме бизнес-цели проекта и пытаться свести список к абсолютному минимуму, необходимому в первом выпуске. Далее, после начала реализации этого выпуска, можно вернуться к ранее исключенным требованиям и повторить весь процесс снова для следующего выпуска.

Трехуровневая шкала приоритетов. Обычный метод расстановки приоритетов подразумевает три группы категорий требований с высоким, средним и низким приоритетами. При этом заинтересованные стороны должны согласовать, что означает каждый уровень в используемой шкале. Один из способов оценки приоритетов предлагает учитывать два измерения: важность и срочность. Каждое требование считается важным либо не важным для достижения бизнес-целей и срочным либо не срочным. Как показано на рисунке 4.4, возможны четыре комбинации для определения шкалы приоритетов.

1. Требования с высоким приоритетом (*high priority*) — и важные (пользователям нужны функции), и срочные (они необходимы уже в следующем выпуске). Некоторые требования приходится включать в эту категорию согласно контрактным или юридическим обязательствам либо из-за непреодолимых бизнес-причин. Если реализацию требования можно отложить до более позднего выпуска без отрицательных последствий, такое требование нельзя считать высокоприоритетным.
2. Требования со средним приоритетом (*medium priority*) — важные (пользователям нужны функции), но не срочные (они могут ждать следующего выпуска).
3. Требования с низким приоритетом (*low priority*) — не важные (пользователи при необходимости могут обойтись без этой функций) и не срочные (пользователи могут ждать, возможно, неограниченно долго).
4. Требования в четвертой клетке кажутся срочными части заинтересованных лиц, возможно, по каким-то «политическим» причинам, но в действительности они не важны для достижения бизнес-целей. Не следует тратить время на работу над ними, поскольку они не добавляют продукту ценности. Если они не важны, следует присвоить им низкий приоритет или исключить из списка требований полностью [23].

	Важно	Не очень важно
Срочно	Высокий приоритет	Это делать не надо!
Не очень срочно	Средний приоритет	Низкий приоритет

Рис. 4.4 – Определение приоритета требований по важности и срочности

При выполнении анализа приоритетов по трехуровневой шкале нужно помнить о зависимостях требований. Высокоприоритетные требования не должны зависеть от требований с более низким приоритетом, запланированных к реализации позже.

100 долларов. Определение приоритетов подразумевает выделение ограниченных ресурсов для достижения максимальной выгоды от средств, которые организация потратила на проект. Один из способов — сделать расстановку приоритетов более наглядной — перевести ее в область более понятных ресурсов, то есть денег. В данном случае речь идет о мнимых деньгах, но все равно это деньги. Дайте команде, занимающейся определением приоритетов, сто воображаемых долларов. Члены команды должны потратить эти доллары на «покупку» конкретных требований из всего набора требований-кандидатов. На более приоритетные требования надо выделить больше денег. Если одно требование в три раза важнее для заинтересованного лица, чем другое, оно должно выделить, скажем, десять долларов на первое и три — на второе. Но каждый получает только 100 долларов — когда деньги кончаются, ничего больше нельзя реализовать, по крайней мере не в том выпуске,

над которым сейчас идет работа. Один способ заключается в привлечении к процессу приоритезации различных участников для распределения своих долларов, после чего суммируются доллары по каждому требованию, чтобы определить, за какие требования было отдано больше всего долларов.

Существуют и другие методики, такие как *MoSCoW* и *Определение приоритетов на основе ценности, стоимости и риска*. Подробнее ознакомиться с ними Вы можете в [23].

4.2.3 Управление требованиями



.....
Под управлением требованиями подразумевают все действия по обеспечению целостности, точности и своевременности обновления соглашения о требованиях в ходе проекта.

На рисунке 4.5 показаны базовые операции по управлению требованиями в четырех основных категориях: управление версиями, управление изменениями, отслеживание состояния требований и отслеживание связей требований.

Управление требованиями			
Управление версиями	Управление изменениями	Отслеживание состояния требований	Отслеживание связей требований
• Определение схемы идентификации версий • Отслеживание версий отдельных требований • Отслеживание версий наборов требований	• Предложение изменений • Анализ влияния изменений • Принятие решений • Обновление отдельных требований • Обновление наборов требований • Обновление планов • Оценка изменчивости требований	• Определение возможных состояний требований • Фиксирование состояния каждого требования • Отслеживание состояния распределения всех требований	• Определение связей с другими требованиями • Определение связей с другими элементами системы

Рис. 4.5 – Основные составляющие управления требованиями

Согласно [28] в ходе управления требованиями нужно решить ряд вопросов:

- Распознавание и учет требований. Каждое требование должно быть индивидуально учтено, поскольку оно может пересекаться с другими требованиями и использоваться в оценках трассировки.
- Управление внесением изменений. Должна предусматриваться последовательность защитных действий для оценки воздействия изменения и стоимости изменения.

- Стратегия трассировки. Существуют зависимости между требованиями, а также между требованиями и проектными решениями системы. Трассировка должна обнаруживать зависимые требования, запоминать эти зависимости и отслеживать влияние требований друг на друга и на проектные решения.

Опишем основные шаги процесса управления изменениями [28].

Шаг 1. Распознавание проблемы. Фиксируется проблема в требованиях или прямой запрос на внесение изменения. Проверяется обоснованность проблемы или запроса. Если обоснованность подтверждена, переходят к следующему шагу. В противном случае процесс прекращается.

Шаг 2. Анализ изменения. При определении возможности изменения исходят из информации трассировки и общих представлений о требованиях к системе. Стоимость изменения определяется двумя параметрами: стоимостью изменения документирования и (если это необходимо) стоимостью изменения проектного решения системы и программного кода. По окончании анализа принимается решение об изменении требования.

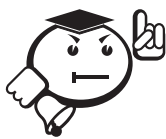
Шаг 3. Выполнение изменения. Вносится изменение в спецификацию требований и, если необходимо, в проектное решение и программный код. Спецификация требований должна быть организована так, чтобы внесение изменения носило локальный характер и не потребовало реорганизации всего документа. Как и в случае программ, изменяемость документов достигается минимизацией внешних ссылок и обеспечением модульности разделов. Это означает, что разделы могут изменяться без влияния на остальные части документа.

Более подробно с процессом управления требованиями можно ознакомиться в [23, 28].

4.3 Как понять, что сбор требований завершен

В общем случае трудно выделить какие-либо признаки, показывающие, что выявление требований завершено. Следующие признаки подскажут вам, что источники сведений уже почти иссякли, по крайней мере на текущем этапе.

- Пользователи уже не могут придумать каких-либо еще вариантов использования или пользовательских историй. Обычно они описывают их в порядке убывания значимости последних.
- Пользователи предлагают новые варианты использования, однако они не приводят к появлению новых функциональных требований. Эти «новые» предложения могут оказаться случаями других вариантов использования, которые вы уже рассмотрели.
- Пользователи повторно описывают уже обсуждавшиеся проблемы.
- Предлагаемые новые функции, пользовательские или функциональные требования выходят за рамки проекта.
- Вновь предлагаемые требования имеют низкий приоритет.
- Пользователи предлагают возможности, которые можно реализовать «когда-то позже», а не включить «в конкретный продукт, который мы сейчас обсуждаем» [23].



.....

Собрать воедино мнения многих пользователей трудно при отсутствии хорошо структурированной организационной схемы, такой, как вариант использования или разделы в спецификации требований. Несмотря на все Ваши усилия собрать все требования, Вам это не удастся, поэтому будьте готовы вносить изменения по мере продвижения. Помните, что Ваша задача — накопить согласованное представление требований, которое достаточно для продолжения работы над следующим выпуском или итерацией при приемлемом уровне риска.

.....

Выявление требований к программному продукту формирует хорошую основу для проектирования системы. В следующей главе пособия мы рассмотрим вопросы проектирования программных систем на основе полученных в ходе сбора требований сведений.



Контрольные вопросы по главе 4

.....

1. Что такое требования в контексте проектирования ПО?
2. Какие типы требований Вам известны?
3. В чем разница между функциональными и нефункциональными требованиями?
4. Что такое бизнес-правила? Назовите основные типы бизнес-правил.
5. Из каких составляющих состоит процесс разработки технических условий?
6. Перечислите основные этапы разработки требований.
7. Назовите основные шаги процесса управления изменениями требований. Как понять, что сбор требований ПО завершен?

Глава 5

МОДЕЛИРОВАНИЕ ТРЕБОВАНИЙ

Визуальные модели требований могут помочь выявить отсутствующие, излишние и несовместимые требования. Ввиду ограничений краткосрочной памяти человека анализ тысячи требований на предмет несоответствий, дублирования и посторонних требований практически невозможен. Вряд ли возможно найти все ошибки путем простого рецензирования текстовых требований [23].

Для создания моделей анализа и проектирования объектно-ориентированных программных систем используют языки визуального моделирования, одним из наиболее популярных из которых на сегодняшний день является UML (*Unified Modeling Language*) — унифицированный язык моделирования.

Спецификация разрабатываемого программного обеспечения при использовании UML объединяет несколько моделей: логическую, использования, реализации, процессов, развертывания.

Логическая модель описывает ключевые понятия моделируемого программного обеспечения (классы, интерфейсы и т. п.), т. е. средства, обеспечивающие его функциональность. *Модель использования* содержит описание функций программного обеспечения с точки зрения пользователя. *Модель реализации* определяет реальную организацию программных модулей в среде разработки. *Модель процессов* отображает организацию вычислений и позволяет оценить производительность, масштабируемость и надежность программного обеспечения. И наконец, *модель развертывания* показывает, каким образом программные компоненты размещаются на конкретном оборудовании.

Все вместе указанные модели, каждая из которых характеризует определенную сторону проектируемого продукта, составляют относительно полную модель разрабатываемого программного обеспечения.

Всего UML предлагает девять дополняющих друг друга диаграмм, входящих в различные модели [28]:

- диаграммы прецедентов (вариантов использования);
- диаграммы классов;

- диаграммы пакетов;
- диаграммы последовательностей действий;
- диаграммы кооперации;
- диаграммы деятельности;
- диаграммы состояний объектов;
- диаграммы компонентов;
- диаграммы размещения.

Данная глава пособия посвящена рассмотрению основных принципов и наиболее важных аспектов использования различных диаграмм UML, необходимых при проектировании программного продукта. Полученные знания помогут Вам описать требуемые модели разрабатываемого программного обеспечения и будут полезны при изучении современных шаблонов архитектуры (глава 6 настоящего пособия).

5.1 Диаграмма прецедентов

Общие сведения

Диаграмма прецедентов (*use case diagram*) относится к концептуальному представлению системы и описывает функциональное предназначение системы. Она служит для определения границ моделируемой системы и описания ее контекста и отражает поведение системы [29]. Именно диаграмма прецедентов, благодаря использованию интуитивно понятной нотации, зачастую используется для организации общения между проектировщиками программного продукта и заказчиками.

Основная идея состоит в представлении системы посредством совокупности прецедентов (вариантов использования, *use cases*) — сервисов, адресованных конкретным потребителям. Любую сущность, взаимодействующую с системой извне и являющуюся потребителем адресованного ей сервиса, называют актером (*actor*). В роли актера может выступать как человек, так и техническое устройство или другая система, служащая источником воздействия на моделируемую систему. С внутренней точки зрения прецедент представляет собой совокупность действий, выполняемых системой в ответ на запрос актера и приводящих к значимому для актера результату.

Следует отметить, что диаграмма прецедентов строится по принципу «черного ящика» — никакой информации о том, каким именно образом реализуется взаимодействие актера и системы, на диаграмме не приводится.



С формальной точки зрения диаграмма прецедентов представляет собой граф, основными элементами которого являются прецеденты, актеры и отношения между ними. С целью упрощения восприятия диаграммы и структурирования информации могут применяться пакеты, содержащие как диаграммы, так и их элементы.

Прецеденты

Прецеденты предназначены для описания поведения моделируемой системы без рассмотрения ее внутренней структуры. Согласно спецификации UML прецеденты обозначаются эллипсом. Имя прецедента, представляющее собой объясняющий назначение прецедента текст, располагается, как правило, непосредственно под эллипсом. Каждый прецедент соответствует отдельному сервису, который моделируемая система предоставляет по запросу пользователя, и содержит законченную последовательность действий, приводящую к значимому для потребителя этого сервиса результату [29].

Именем прецедента должен быть глагол либо отглагольное существительное, обозначающее действие и поясняющие слова. Так, два прецедента, приведенные на рисунке 5.1, считаются эквивалентными.



Рис. 5.1 – Графическая нотация прецедентов

Актеры

Актером в рамках диаграммы прецедентов является любая внешняя по отношению к моделируемой системе сущность, непосредственно взаимодействующая с ней и использующая ее для достижения определенных целей и решения частных задач [29]. Каждый актер обозначает определенную роль, которую может играть элемент внешней среды при взаимодействии с системой.

Стандартным обозначением актера является пиктограмма, приведенная на рисунке 5.1, под которой располагается имя актера.

Имя актера начинается с заглавной буквы. Часто имена актеров совпадают с должностями, занимаемыми сотрудниками в организации. Такая ситуация типична для многопользовательских информационных систем, ориентированных на использование сотрудниками соответствующих подразделений. Однако ключевым принципом в формировании имени актера безусловно является роль, которую он играет по отношению к системе. Из этого вытекает необходимость использовать в качестве имени актера нарицательные существительные, а не имена собственные. Возможными примерами наименований актеров являются *пользователь*, *продавец*, *кассир*, *администратор системы*, *заместитель руководителя предприятия*.

Важно отметить, что актер не является частью моделируемой системы, поэтому его внутренняя структура никак не специфицируется, важна лишь роль актера по отношению к системе [29].

Абстрактные актеры и прецеденты

Правила построения диаграммы прецедентов допускают размещение на ней абстрактных прецедентов и абстрактных актеров.



.....
***Абстрактным** является прецедент, который не может иметь экземпляров.*

Абстрактный прецедент не может быть запущен и, таким образом, не может быть связан с актерами. Такие прецеденты используются для выделения в рамках моделируемой системы некоторой функциональности, которая может быть использована другими прецедентами.

Как правило, абстрактные прецеденты связываются с обычными прецедентами отношением обобщения, как показано на рисунке 5.2. При этом абстрактный прецедент является предком.

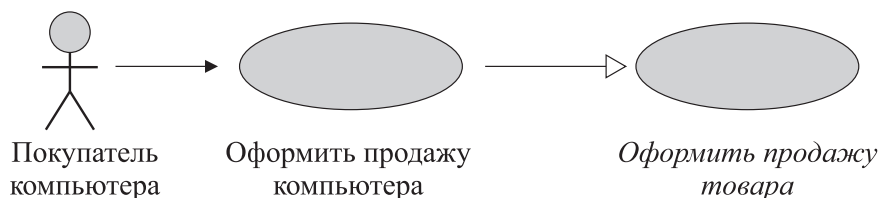


Рис. 5.2 – Пример абстрактного прецедента

Абстрактным актером называется актер, который не может иметь экземпляров. Использование абстрактных актеров и прецедентов соответствует парадигме наследования в рамках объектно-ориентированного анализа и проектирования [30]. Например, в компании может быть несколько разновидностей служащих: служащий, работающий на ставку; служащий с почасовой оплатой; служащий, принятый на испытательный срок. Все актеры имеют общие свойства, фиксируемые на диаграмме путем введения абстрактного актера «Служащий» и указания отношения между ними, как показано на рисунке 5.3.

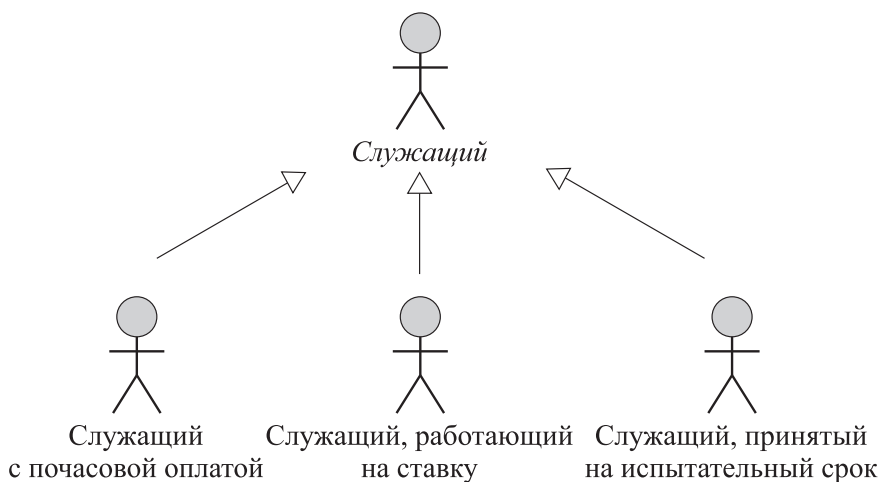


Рис. 5.3 – Пример абстрактного актера

Абстрактные актеры не могут иметь экземпляров, а значит, не могут инициировать прецеденты.



Абстрактные актеры не могут быть связаны с прецедентами.

Отношение ассоциации

Язык UML определяет несколько типов отношений для описания взаимодействия различных элементов диаграммы. Основными типами используемых отношений диаграммы прецедентов являются отношения ассоциации (*association relationship*), расширения (*extend relationship*), включения (*include relationship*) и обобщения (*generalization relationship*).

Отношение ассоциации используется в той или иной форме при создании всех канонических диаграмм языка [29]. В общем случае оно обозначает наличие некоторого «знания» у связанных элементов друг о друге. Если ассоциация является направленной, то элемент, от которого отношение исходит, «знает» об элементе, к которому отношение направлено.

Отношение ассоциации изображается на диаграммах сплошной линией и дополнительно может характеризоваться направлением, именем и кратностью.

Направление ассоциации призвано показать лицу, читающему диаграмму, кто является инициатором взаимодействия: актер или система. Многие разработчики диаграмм не приводят подобной информации и предпочитают использовать ненаправленные ассоциации (*association*). Можно сказать, что диаграммы, приведенные на рисунке 5.4, эквивалентны, однако верхняя диаграмма более информативна, поскольку, глядя на нее, можно сказать, что некоторая информационная система предоставляет бухгалтерской системе данные о сотрудниках на основании запроса последней. Инициатором взаимодействия в данном примере является актер.

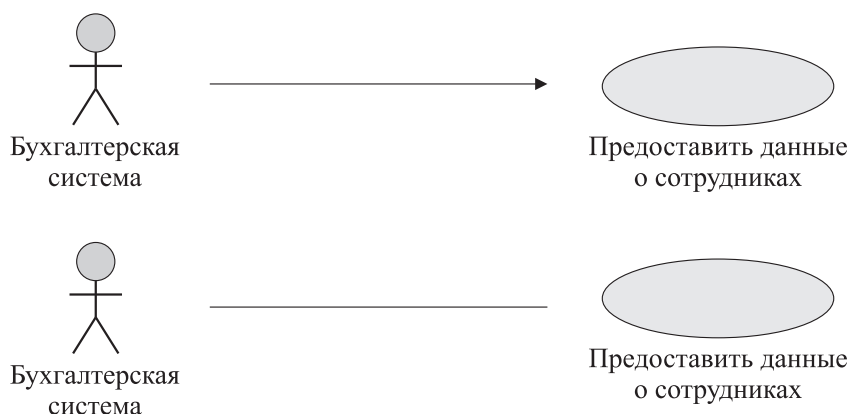


Рис. 5.4 – Направленные и ненаправленные ассоциации

Кратность (*multiplicity*) ассоциации может быть указана на обоих концах отношения и указывает на количество экземпляров элементов, участвующих в отношении. Так, на рисунке 5.5 кратность, указанная в форме «*», означает, что каждый отдельный клиент может оформить на себя несколько кредитов. При этом их число не известно, заранее ничем не ограничено и может быть равно 0, т. е. некоторые клиенты могут не брать кредитов вовсе.



Рис. 5.5 – Кратность отношения ассоциации

На другом конце отношения кратность указана равной единице. Это означает, что отдельно взятый кредит может оформляться на одного и только одного клиента банка.



.....
Если кратность отношения ассоциации не указана, она по умолчанию принимается равной единице.
.....

Отношение расширения

Отношение расширения всегда является направленным и образуется путем наложения стереотипа «*extend*» («расширяет») на отношение зависимости. Его формальное определение выглядит следующим образом.



.....
Отношение расширения, направленное от прецедента *A* к прецеденту *B*, означает, что свойства экземпляра прецедента *B* могут быть дополнены свойствами расширяющего прецедента *A*.
.....

Отношение изображается пунктирной стрелкой и направлено от прецедента, расширяющего исходный прецедент, как показано на рисунке 5.6.

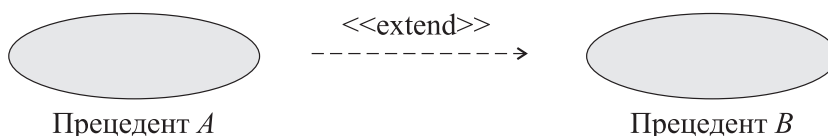


Рис. 5.6 – Отношение расширения

Отношение расширения служит для обозначения того факта, что один из прецедентов может присоединить к своему поведению некоторое дополнительное поведение, определенное для другого прецедента. Предполагается, что существует некоторое условие и место, называемое *точкой расширения*, где в поведение базового прецедента включается поведение расширяющего его прецедента, если это условие истинно.

Один прецедент может быть расширяющим для нескольких прецедентов и, в свою очередь, иметь в качестве собственных расширений несколько других прецедентов.

На рисунке 5.7 приведен пример использования отношения расширения.

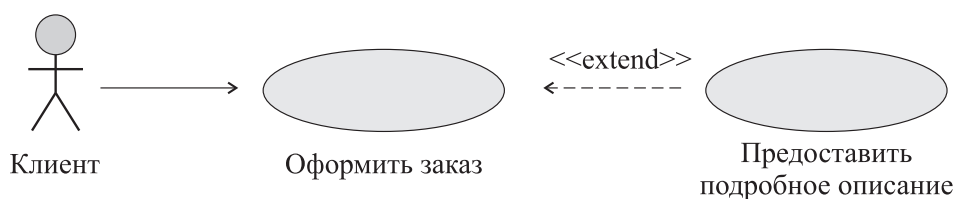


Рис. 5.7 – Пример использования отношения расширения

Для определения точек расширения в форме булевых выражений используются описания потоков событий, которые будут рассмотрены позже.

Отношение включения

Отношение включения, как и отношение расширения, всегда является направленным и образуется путем наложения стереотипа «*include*» («включает») на отношение зависимости. Его формальное определение выглядит следующим образом.



.....
Отношение включения, направленное от прецедента *A* к прецеденту *B*, указывает, что каждый экземпляр прецедента *A* включает в себя функциональные свойства прецедента *B*.

Отношение изображается пунктирной стрелкой и направлено к прецеденту, включаемому в более общий, как показано на рисунке 5.8.

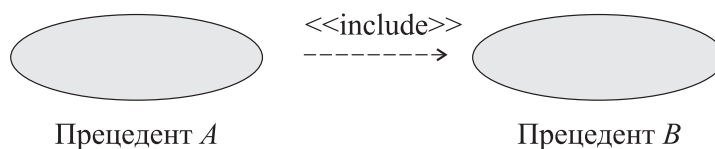


Рис. 5.8 – Отношение включения

Отношения включения между двумя прецедентами указывают, что некоторое поведение одного прецедента включается в качестве составного компонента в последовательность действий другого прецедента, называемого базовым. При этом базовый прецедент может зависеть от результатов выполнения включаемого в него прецедента.

Пример использования отношения включения приведен на рисунке 5.9.



Рис. 5.9 – Пример использования отношения включения

Отношение обобщения

Отношение обобщения (*generalization relationship*) указывает, что некоторый прецедент *A* может быть обобщен до прецедента *B*. В этом случае прецедент *A* будет являться специализацией или потомком прецедента *B*, а *B* будет считаться предком или родителем прецедента *A*. Потомок наследует все свойства и поведение родителя и может быть дополнен новыми свойствами и особенностями поведения.

На диаграмме отношение обобщения представляется сплошной стрелкой, на конце которой располагается незакрашенный треугольник. Отношение обобщения всегда является направленным, указывая на родительский элемент, как показано на рисунке 5.10.

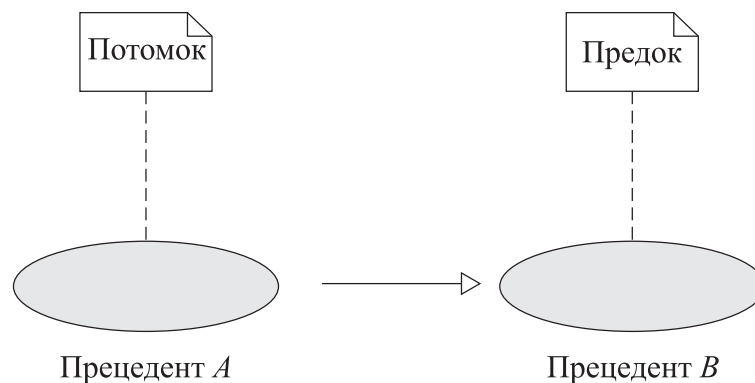


Рис. 5.10 – Отношение обобщения между прецедентами

На практике отношение обобщения применяют, когда необходимо показать, что дочерние прецеденты обладают всеми свойствами и особенностями поведения родительских прецедентов [29]. Один дочерний прецедент может быть потомком нескольких родительских. Тогда между прецедентами возникает множественное наследование [30], и дочерний прецедент наделяется свойствами каждого из прецедентов-предков. Возможна и обратная ситуация, когда один родительский прецедент порождает несколько прецедентов-потомков.

Пример использования отношения обобщения между прецедентами приведен на рисунке 5.11.

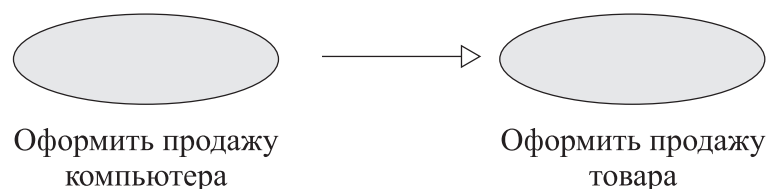


Рис. 5.11 – Пример использования отношения обобщения между прецедентами

Следует отметить, что отношение обобщения может возникать и между актерами, как показано на рисунке 5.12.

Отношение обобщения, направленное от актера *A* к актеру *B*, указывает, что каждый экземпляр актера *A* является одновременно экземпляром актера *B* и обладает всеми его свойствами. Актер *B* в таком случае является родителем по отношению к актеру *A*, а актер *A* — его потомком.

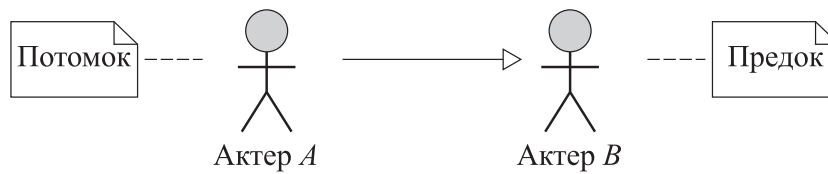


Рис. 5.12 – Отношение обобщения между актерами

Потоки событий

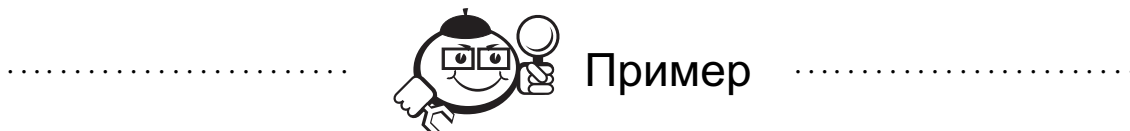
Прецеденты описывают поведение создаваемой системы с точки зрения актера, т.е. точки зрения, внешней по отношению к объекту моделирования. Структура прецедента определяется в документе, называемом «поток событий» (*flow of events*) и являющемся частью его спецификации. Целью создания документа «поток событий» является документирование процесса обработки данных, реализуемого в рамках прецедента. Этот документ подробно описывает действия пользователей и описки системы.

Обычно поток событий содержит:

- краткое описание;
- предусловия (*pre-conditions*);
- описание того, каким образом запускается прецедент;
- основной поток событий (типичный ход событий);
- альтернативные потоки событий;
- потоки ошибок;
- описание того, каким образом завершается прецедент;
- постусловия (*post-conditions*).

Предусловия прецедента представляют собой условия, которые должны быть выполнены, прежде чем прецедент начнет выполняться. Например, в роли предусловия может быть указана необходимость выполнения другого прецедента или наличие у пользователя прав, требуемых для запуска прецедента. Диаграммы прецедентов не содержат информации о порядке их выполнения. Использование предусловия предоставляет возможность документировать эти сведения в форме высказываний на естественном языке. Наличие предусловий для прецедента не является обязательным.

Постусловиями называются условия, которые должны быть выполнены после завершения прецедента. Правила формулирования пред- и постусловий — идентичны.



Ниже приведен поток событий прецедента «Снять наличные». Рассматривается процесс взаимодействия пользователя и банкомата.

Основной поток событий.

Выполнение прецедента начинается, когда пользователь вставляет свою карточку в банкомат.

Банкомат выдает приветствие и предлагает пользователю ввести свой персональный идентификационный номер (код пластиковой карточки).

Клиент вводит номер.

Банкомат проверяет введенный код. Если код не подтверждается, выполняется альтернативный поток событий A1.

Банкомат выводит список доступных действий (снять наличные, получить распечатку, проверить остаток и т. д.)

Пользователь выбирает пункт «Снять наличные».

Банкомат запрашивает, сколько денег нужно снять.

Пользователь вводит требуемую сумму.

Банкомат определяет, достаточно ли на счету денег. Если денег недостаточно, выполняется альтернативный поток A2. Если во время подтверждения суммы возникают ошибки, выполняется поток ошибок E1.

Банкомат вычитает требуемую сумму из суммы на счету пользователя.

Банкомат выдает пользователю требуемую сумму наличными.

Банкомат возвращает пользователю его карточку.

Прецедент завершается.

Альтернативный поток A1: ввод неправильного кода.

Банкомат информирует пользователя, что код введен неправильно.

Банкомат возвращает пользователю его карточку.

Прецедент завершается.

Альтернативный поток A2: недостаточно денег на счету.

Банкомат информирует пользователя, что денег на его счету недостаточно.

Банкомат возвращает пользователю его карточку.

Прецедент завершается.

Поток ошибок E1: ошибка при подтверждении суммы.

Банкомат сообщает пользователю, что при подтверждении запрашиваемой суммы произошла ошибка.

Банкомат заносит сведения об ошибке в журнал ошибок. Каждая запись содержит дату и время ошибки, имя клиента, номер его счета и код ошибки.

Банкомат возвращает пользователю его карточку.

Прецедент завершается.

.....

В зависимости от описания потока событий прецедент может быть идеальным (*essential use cases*) или реальным (*real use cases*). На диаграммах такие прецеденты ничем не отличаются друг от друга. Один и тот же прецедент может являться и идеальным, и реальным — все зависит от того, каким образом описан поток событий прецедента.



.....
Идеальные прецеденты — это прецеденты, выражающие общую сущность процесса без детализации его протекания.

Проектные решения, особенно связанные с интерфейсом, в описании таких прецедентов опускаются. Идеальный прецедент описывает процесс в терминах,

наиболее важных видов деятельности актера и откликов на них со стороны системы. Степень абстракции идеальных прецедентов может варьироваться, т. е. прецедент может быть идеальным в большей или меньшей степени.

В рамках идеального потока событий для прецедента «Снятие наличных» возможна формулировка, предложенная в таблице 5.1.

Таблица 5.1 – Формулировка действий и откликов в идеальном прецеденте

Действия актера	Отклик системы
Пользователь идентифицирует себя	Предоставляет перечень возможностей



.....
***Реальные прецеденты** — это прецеденты, которые конкретно описывают процесс в терминах проектных решений, на основе конкретных технологий ввода-вывода информации.*

Когда речь заходит об интерфейсе пользователя, реальные прецеденты зачастую определяют содержимое диалоговых окон и описывают способы взаимодействия с конкретными устройствами. В рамках реального потока событий для прецедента «Снятие наличных» возможна формулировка, предложенная в таблице 5.2.

Таблица 5.2 – Формулировка действий и откликов в реальном прецеденте

Действия актера	Отклик системы
Пользователь вставляет свою пластиковую карточку	Выдает приглашение на ввод кода
Вводит пароль с использованием клавиатуры	Отображает пункты меню, доступные для использования
И т. д.	И т. д.

В заключение отметим, что очень важно отличать прецеденты от функций системы. Для иллюстрации обратимся к статье Курта Биттнера (Kurt Bittner) — топ-менеджера компании Rational Software, в которой автор говорит о том, что прецедент следует рассматривать как описание того, как можно использовать систему, чтобы получить *нечто полезное* [6]. Таким образом, в случае Интернет-магазина множество таких прецедентов, как «Получить сведения о заказах», «Оформить заказ», «Добавить заказ» и «Удалить заказ», не представляющих ценности для клиента по отдельности, можно заменить одним вариантом использования «Управление заказами».

5.2 Диаграмма состояний

Общие сведения

Каждая информационная система характеризуется некоторым поведением и функциональностью. Рассмотренные выше диаграммы прецедентов предназначены для описания сервисов, которые предоставляет актерам моделируемая система. Для моделирования поведения на логическом уровне в языке UML могут использоваться сразу несколько канонических диаграмм: состояний, деятельности, последовательности и кооперации, каждая из которых фиксирует внимание на отдельном аспекте функционирования системы. В отличие от других диаграмм диаграмма состояний (*statechart diagram*) описывает процесс изменения состояний только одного объекта. При этом изменение состояния объекта может быть вызвано внешними воздействиями со стороны других объектов или извне. В данном случае термин «объект» трактуется шире, чем это принято в объектно-ориентированном анализе и проектировании (ООАП), который определяет его как экземпляр некоторого класса. В роли объекта может выступать система в целом, ее часть, отдельные компоненты и даже актеры [29].

Диаграмма состояний представляет динамическое поведение сущностей на основе спецификации их реакции на восприятие некоторых конкретных событий.

Диаграмма состояний — это представление конечного автомата в виде графа, вершинами которого являются состояния, а дугами — переходы между ними. Диаграммы состояний могут быть вложены друг в друга, образуя вложенные диаграммы. Более подробно ознакомиться со вложенными диаграммами состояний можно в [31]. Для понимания семантики конкретной диаграммы состояний, в отличие от диаграмм прецедентов, необходимо представлять не только особенности поведения моделируемой сущности, но и владеть общими сведениями теории автоматов.

Автоматы

Автомат (*state machine*) описывает поведение объекта в форме последовательности состояний, которые охватывают все этапы его жизненного цикла, начиная от создания объекта и заканчивая его уничтожением. Каждая диаграмма состояний представляет некоторый автомат.

Основными понятиями, входящими в формализм автомата, являются «состояние» и «переход». Главное различие между ними заключается в том, что длительность нахождения системы в отдельном состоянии существенно превышает время, которое затрачивается на переход из одного состояния в другое. Предполагается, что в пределе время перехода из одного состояния в другое равно нулю (если дополнительно об этом ничего не сказано). Другими словами, переход объекта из состояния в состояние происходит мгновенно.

Поведение системы в случае диаграммы состояний моделируется как последовательное перемещение по графу состояний от вершины к вершине по связывающим их дугам с учетом их ориентации.

Граф состояний является конечным [31], то есть содержит конечное число состояний. Кроме того, граф состояний обладает специальными свойствами. Одним

из таких свойств является выделение из всей совокупности состояний двух специальных: начального и конечного. Хотя на диаграмме состояний время нахождения системы в том или ином состоянии явно не учитывается, предполагается, что последовательность изменения состояний упорядочена во времени. Другими словами, каждое последующее состояние всегда наступает позже предшествующего ему состояния.

Еще одним свойством графа состояний является достижимость состояний. Речь идет о том, что навигация или ориентированный путь в графе состояний определяет специальное бинарное отношение на множестве всех состояний системы. Это отношение характеризует потенциальную возможность перехода системы из рассматриваемого состояния в некоторое другое состояние. Очевидно, для достижимости состояний необходимо наличие связывающего их ориентированного пути в графе состояний.

Формализм обычного автомата основан на выполнении следующих обязательных условий [29].

1. Автомат не запоминает историю перемещения из состояния в состояние.
2. В каждый момент времени автомат может находиться в одном и только в одном из своих состояний.
3. Хотя процесс изменения состояний автомата происходит во времени, явно концепция времени не входит в формализм автомата.
4. Граф автомата не должен содержать изолированных состояний и переходов. Это условие означает, что для каждого из состояний, кроме начального, должно быть определено предшествующее состояние. Каждый переход должен обязательно соединять два состояния автомата. Допускается переход из состояния в себя, такой переход еще называют петлей.
5. Автомат не должен содержать конфликтующих переходов, т. е. таких переходов из одного и того же состояния, когда объект одновременно может перейти в два и более последующих состояния.

Состояние

В языке UML под состоянием (*state*) понимается элемент модели, используемый для фиксации отдельной ситуации, в течение которой имеет место выполнение некоторого условия [29]. Согласно [32] состояние представляет собой абстракцию значений и связей объекта. Состояние позволяет зафиксировать ситуацию, когда объект находится в ожидании возникновения некоторого внешнего события, и в то же время состояние используется для моделирования динамических аспектов, когда в период времени, в который оно имеет место, выполняются некоторые действия. В этом случае моделируемый элемент переходит в рассматриваемое состояние в момент начала соответствующей деятельности и покидает данное состояние в момент ее завершения.

Состояние на диаграмме изображается прямоугольником со скругленными вершинами, как показано на рисунке 5.13. Этот прямоугольник может быть разделен на две секции горизонтальной линией. В верхней секции указывается имя состояния, а в нижней — список внутренних действий. Нижняя секция может отсутствовать.

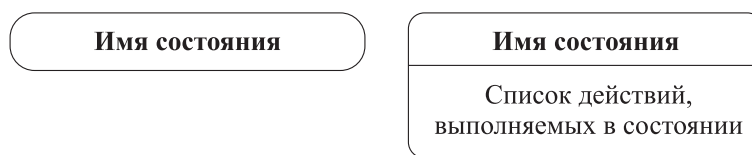


Рис. 5.13 – Графическое изображение состояний на диаграмме

Имя состояния представляет собой строку текста, которая раскрывает содержательный смысл данного состояния. Имя всегда записывается с заглавной буквы.

Секция списка внутренних действий содержит перечень внутренних действий или деятельности, которые выполняются в процессе нахождения моделируемого элемента в данном состоянии. Каждое из действий записывается в виде отдельной строки и имеет формат *<метка действия>/'<выражение действия>*.

Метка действия указывает на обстоятельства или условия, при которых будет выполняться действие, определенное выражением действия. В UML для метки действия имеются три предопределенных значения:

- *entry* — эта метка указывает на действие, которое выполняется в момент входа в данное состояние (входное действие);
- *exit* — эта метка указывает на действие, которое выполняется в момент выхода из данного состояния (выходное действие);
- *do* — эта метка специфицирует деятельность («*do activity*»), которая выполняется в течение всего времени, пока объект находится в данном состоянии.

Разработчик вправе вводить собственные метки действия, придерживаясь следующего правила: метка действия идентифицирует событие, которое запускает расположенное после нее выражение действия.

Начальное и конечное состояния



.....
Начальное состояние представляет собой частный случай состояния, которое не содержит никаких внутренних действий. В этом состоянии находится объект по умолчанию в начальный момент времени.

Оно служит для указания на диаграмме состояний графической области, от которой начинается процесс изменения состояний. Графически начальное состояние в языке UML обозначается в виде заливной окружности, из которой может только выходить стрелка, соответствующая переходу.

Переход из начального состояния может быть помечен событием создания (*инициализации*) объекта. В противном случае переход никак не помечается.



.....
Конечное (финальное) состояние представляет собой частный случай состояния, которое также не может содержать никаких внутренних действий. В этом состоянии будет находиться объект по умолчанию после завершения работы автомата в конечный момент времени.

Оно служит для указания на диаграмме состояний графической области, в которой завершается процесс изменения состояний или жизненный цикл данного объекта. Графически конечное состояние в языке UML обозначается в виде заливной окружности, помещенной в окружность большего диаметра. Все переходы для конечного состояния могут быть только входящими.

Переход

Переход (*transition*) — это мгновенная смена одного состояния другим. Пребывание моделируемого объекта в первом состоянии может сопровождаться выполнением некоторых действий, а переход во второе состояние будет возможен после завершения этих действий, а также после удовлетворения некоторых дополнительных условий [29]. В этом случае говорят, что переход срабатывает, или происходит срабатывание перехода.

Каждый объект класса диаграммы UML может иметь конечное число состояний и в один момент времени находиться только в одном из них [32].

Состояние, в котором объект находится до срабатывания перехода, называется *исходным*, а состояние, в которое объект переходит в результате срабатывания, — *целевым*.

На переходе могут указываться события, вызывающие его срабатывание, и действия, производимые объектом при переходе из одного состояния в другое. Срабатывание перехода может зависеть не только от наступления некоторого события, но и от выполнения определенного условия, называемого сторожевым. При этом объект перейдет из одного состояния в другое в том случае, если произошло указанное событие и сторожевое условие приняло значение «истина».

На диаграмме состояний переход изображается сплошной линией со стрелкой, которая направлена в целевое состояние. Если целевое и исходное состояния перехода совпадают (такой переход называют *переходом в себя*), переход изображается петлей со стрелкой. Следует отметить, что при переходе в себя объект покидает исходное состояние, а затем снова входит в него. Таким образом, при переходе в себя выполняются действия, специфицированные метками *entry* и *exit*.

Каждый переход может быть помечен строкой текста, которая имеет следующий общий формат: <сигнатура события>'/'<сторожевое условие>'/'<выражение действия>.

В свою очередь сигнатура события описывает некоторое событие с необходимыми аргументами: <имя события>'('<список параметров>')'.

В языке UML события играют роль стимулов, которые инициируют переходы из одних состояний в другие. Событие — это происшествие, которое случилось

в определенный момент времени. В рамках UML выделяют события сигнала, события изменения и события времени. *Имя события* используется для идентификации отдельных переходов на диаграмме состояний и представляет собой строку текста, начинающуюся со строчной буквы. Переходы с указанными именами событий называются триггерными.

Если рядом со стрелкой перехода не указана никакая строка текста, то соответствующий переход является нетриггерным, и в этом случае из контекста диаграммы состояний должно быть ясно, после окончания какой деятельности он срабатывает. После имени события могут следовать круглые скобки для явного задания параметров соответствующего события-триггера. Если таких параметров нет, то список параметров со скобками может отсутствовать.

В общем случае из одного состояния может быть несколько переходов с одним и тем же событием-триггером. При этом никакие два сторожевых условия не должны одновременно принимать значение «истина». Каждое из сторожевых условий необходимо вычислять всякий раз при наступлении соответствующего события-триггера.

Выражение действия (*action expression*) выполняется в том и только в том случае, когда переход срабатывает. Выражение действия представляет собой атомарную операцию (достаточно простое вычисление), выполняемую сразу после срабатывания соответствующего перехода до начала каких бы то ни было действий в целевом состоянии. Атомарность действия означает, что оно не может быть прервано никаким другим действием до тех пор, пока не закончится его выполнение. Данное действие может оказывать влияние как на сам объект, так и на его окружение, если это с очевидностью следует из контекста модели. Выражение записывается после знака «/» в строке текста, присоединенной к соответствующему переходу.

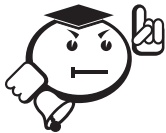
5.3 Диаграмма деятельности

Общие сведения

При моделировании поведения проектируемой или анализируемой системы возникает необходимость не только представить процесс изменения ее состояний, но и детализировать особенности алгоритмической и логической реализации выполняемых системой операций [29].

Для моделирования процесса выполнения операций в языке UML используются диаграммы деятельности. Диаграммы деятельности показывают последовательность этапов, образующих некоторый сложный процесс [32]. Применяемая в них графическая нотация во многом похожа на нотацию диаграммы состояний, поскольку на диаграммах деятельности также присутствуют обозначения состояний и переходов. Отличие заключается в семантике состояний, которые используются для представления не деятельностей, а действий, и в отсутствии на переходах сигнатуры событий. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, и переход в следующее состояние срабатывает только при ее завершении. Графически диаграмма деятельности представляется в форме графа, вершинами которого являются состояния действия,

а дугами — переходы от одного состояния действия к другому. Таким образом, диаграммы деятельности можно считать частным случаем диаграмм состояний.



.....
 Диаграмма деятельности похожа на обычную блок-схему, она, как и блок-схема, показывает поток управления от этапа к этапу. Однако, в отличие от блок-схемы, диаграмма деятельности может отражать как последовательную, так и параллельную деятельность [32].

Состояние действия и состояние деятельности

Состояние действия (*action state*) является специальным случаем состояния с некоторым входным действием и, по крайней мере, одним выходящим из состояния переходом. Обычное использование состояния действия заключается в моделировании одного шага выполнения алгоритма (процедуры) или потока управления.

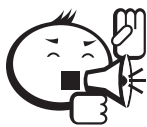
Графически состояние действия изображается фигурой, напоминающей прямоугольник, боковые стороны которого заменены выпуклыми дугами, как показано на рисунке 5.14. Внутри этой фигуры записывается выражение действия (*action expression*), которое должно быть уникальным в пределах одной диаграммы деятельности.

Разобрать план проекта

$N := N + 1$

Рис. 5.14 – Графическое изображение состояния действия

Действие может быть записано на естественном языке, псевдокоде или языке программирования. Никаких дополнительных или неявных ограничений при записи действий не накладывается [29].



.....
 Состояния действия не могут быть подвергнуты декомпозиции, поскольку они являются атомарными. Это значит, что в период пребывания системы в состоянии действия выполняемая работа не может быть прервана. Обычно предполагается, что длительность пребывания системы в состоянии действия занимает неощутимо малое время.

В отличие от состояния действия состояния деятельности могут быть декомпозированы для более детального представления данной деятельности. Состояния деятельности не являются атомарными, то есть могут быть прерваны. Предполагается, что для их завершения требуется заметное время [29, 32]. Можно считать, что состояние действия — это частный вид состояния деятельности, а конкретнее — такое состояние, которое не может быть подвергнуто дальнейшей декомпозиции. Состояние деятельности, в свою очередь, можно представлять себе как составное состояние, поток управления которого включает только другие состояния деятельности и действий (рис. 5.15).

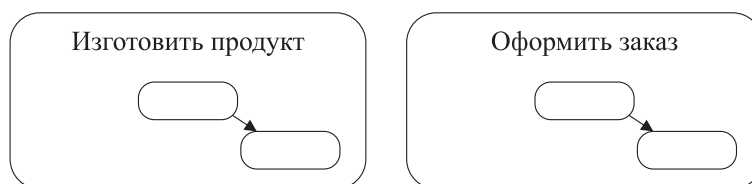


Рис. 5.15 – Графическое изображение состояния деятельности, содержащего вложенные действия

Каждая диаграмма деятельности должна иметь единственное начальное и единственное конечное состояния. Они имеют такие же обозначения, как и на диаграмме состояний. Каждая деятельность начинается в начальном состоянии и заканчивается в конечном состоянии. Диаграмму деятельности, как правило, располагают таким образом, чтобы действия следовали сверху вниз или слева направо.

Переход

Переход как элемент языка UML был рассмотрен при изучении диаграммы состояний (разд. 5.2). При построении диаграммы деятельности используются только нетриггерные переходы, т.е. такие, которые срабатывают сразу после завершения деятельности или выполнения соответствующего действия. Нетриггерный переход переводит систему в последующее состояние сразу, как только закончится действие в предыдущем состоянии. На диаграмме такой переход изображается сплошной линией со стрелкой.

Если из состояния действия выходит единственный переход, то он может быть никак не помечен. Если же таких переходов несколько, то сработать может только один из них. Именно в этом случае для каждого из таких переходов должно быть явно записано сторожевое условие в прямых скобках.

Ветвление

Графически ветвление на диаграмме деятельности обозначается небольшим ромбом, внутри которого нет никакого текста. В этот ромб может входить только одна стрелка от того состояния действия, после выполнения которого поток управления должен быть продолжен по одной из взаимно исключающих ветвей. Принято входящую стрелку присоединять к верхней или левой вершине символа ветвления. Выходящих стрелок может быть две или более, но для каждой из них явно указывается соответствующее сторожевое условие в форме булевского выражения.



Рассмотрим диаграмму, приведенную на рисунке 5.16. Моделируется ситуация расчета общей стоимости товаров, покупаемых по кредитной карточке в супермаркете. Если общая стоимость превышает \$50, то выполняется аутентификация

личности владельца карточки. В случае положительной проверки (карточка действительна) или если стоимость товаров не превышает \$50, происходит снятие суммы со счета и оплата стоимости товаров. При отрицательном результате (карточка недействительна) оплаты не происходит, и товар остается у продавца.

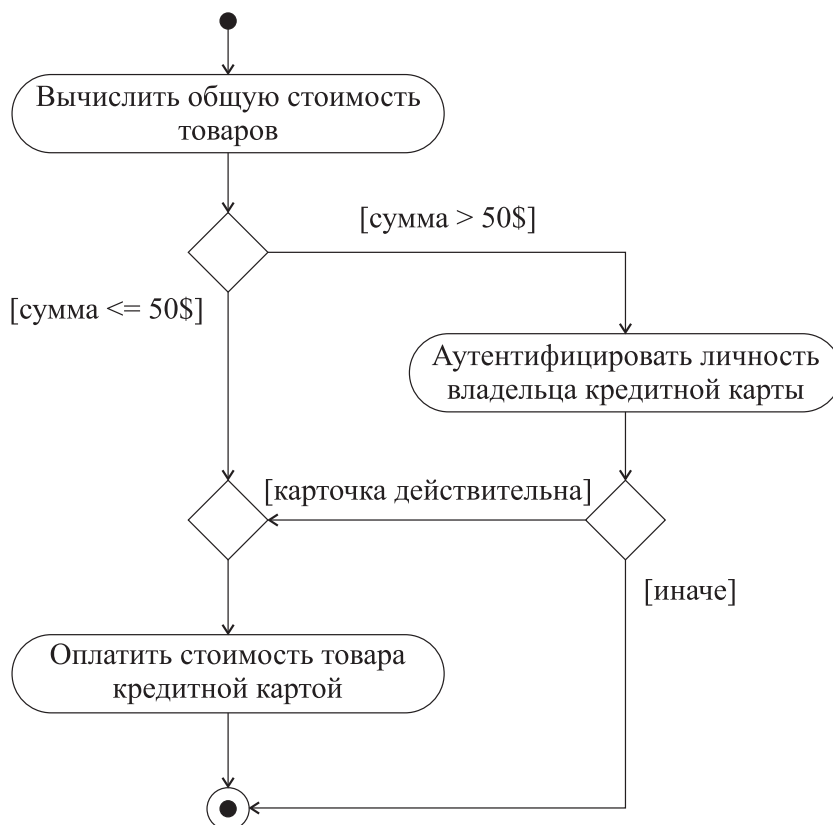


Рис. 5.16 – Пример использования ветвлений на диаграмме деятельности

Разделение и слияние

Один из наиболее значимых недостатков обычных блок-схем или структурных схем алгоритмов связан с проблемой изображения параллельных ветвей отдельных вычислений [29]. Организации и компьютерные системы способны выполнять несколько действий одновременно. В языке UML для разделения и слияния параллельных вычислений или потоков управления используется специальный символ, графической нотацией которого является прямая черта (рис. 5.17), которая, как правило, изображается отрезком горизонтальной линии, толщина которой несколько шире основных линий диаграммы деятельности. При этом разделение (*concurrent fork*) имеет один входящий переход и несколько исходящих. Слияние (*concurrent join*), напротив, имеет несколько входящих переходов и один выходящий, как показано на рисунке 5.17.

Следует обратить внимание на необходимость поддержания баланса между точками разделения и слияния.

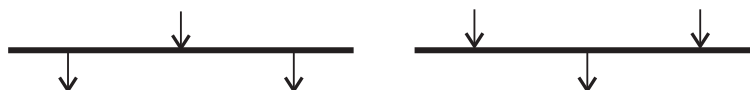


Рис. 5.17 – Изображение разделения и слияния параллельных потоков



.....
 Число потоков, исходящих из точки разделения, должно быть равно числу потоков, приходящих в соответствующую ей точку слияния.

Дорожки

Деятельность любой компании можно представить как совокупность отдельных действий, направленных на достижение требуемого результата. Таким образом, диаграммы деятельности могут быть использованы не только для спецификации алгоритмов вычислений или потоков управления в программных системах, но и для моделирования бизнес-процессов.

При моделировании бизнес-процессов рекомендуется выполнение каждого действия ассоциировать с конкретным подразделением компании [29]. В этом случае подразделение несет ответственность за реализацию отдельных действий, а сам бизнес-процесс представляется в виде переходов действий из одного подразделения к другому.

Для моделирования этих особенностей в языке UML используется специальная конструкция, получившая название дорожки (*swimlanes*). Такое название возникло в силу визуальной аналогии с плавательными дорожками в бассейне. Все состояния действия на диаграмме деятельности делятся на отдельные группы, которые отделяются друг от друга вертикальными линиями. Две соседние линии образуют дорожку, а группа состояний между этими линиями выполняется отдельным подразделением компании.

Названия подразделений явно указываются в верхней части дорожки. Пересекать линию дорожки могут только переходы, которые в этом случае обозначают выход или вход потока управления в соответствующее подразделение компании. Порядок следования дорожек не несет какой-либо семантической информации и определяется соображениями удобства.



Пример

В качестве примера рассмотрим фрагмент диаграммы деятельности торговой компании, приведенный на рисунке 5.18. Подразделениями компании являются отдел обслуживания клиентов, отдел продаж и склад.

Этим подразделениям будут соответствовать три дорожки на диаграмме деятельности, каждая из которых специфицирует зону ответственности подразделения. В данном случае диаграмма деятельности включает в себе не только информацию о последовательности выполнения действий, но и о том, какое из подразделений компании должно выполнять закрепленное за ним действие.

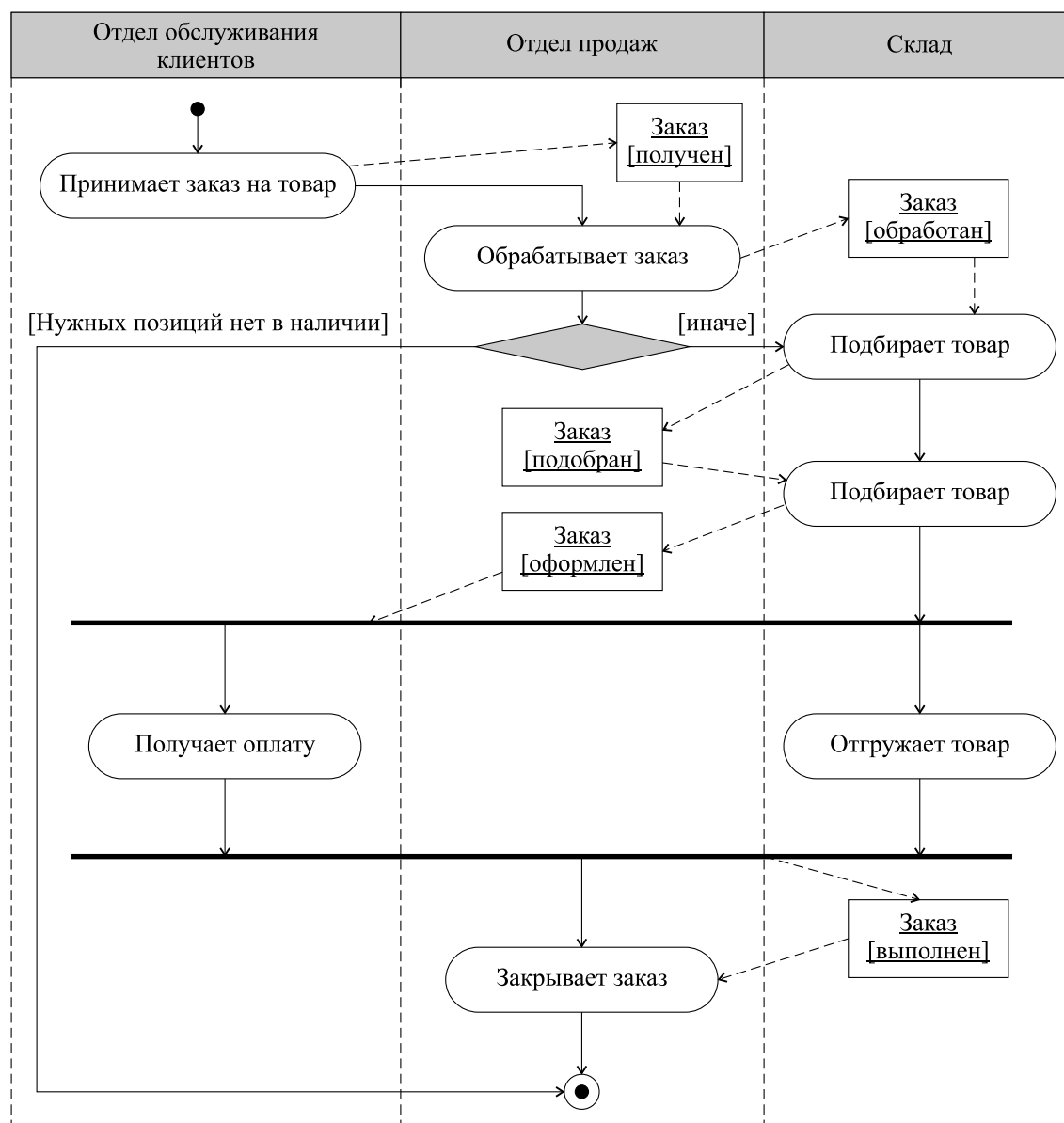


Рис. 5.18 – Фрагмент диаграммы деятельности торговой компании

5.4 Диаграмма классов

Общие сведения

Центральное место в ООАП занимает разработка логической модели системы в виде диаграммы классов. Нотация классов в языке UML отличается простотой и, как правило, интуитивно понятна разработчикам, имеющим опыт работы с CASE-инструментариями. Схожая нотация применяется и для объектов — экземпляров класса, с тем различием, что к имени класса добавляется имя объекта и вся надпись подчеркивается.

Диаграмма классов (*class diagram*) служит для представления статической структуры модели системы в терминологии классов объектно-ориентированного программирования. Диаграмма классов отражает различные взаимосвязи между отдельными сущностями предметной области, а также описывает их внутреннюю структуру и типы отношений. В то же время диаграмма классов не содержит сведений о временных аспектах функционирования системы.

Диаграмма представляет собой граф, вершинами которого являются элементы типа «классификатор», которые связаны различными типами структурных отношений. Элементами диаграммы классов являются классы, интерфейсы, пакеты, объекты и связи.

Класс

Класс (*class*) в языке UML служит для обозначения множества объектов, которые обладают идентичной структурой, поведением и отношениями с объектами из других классов. Графически класс изображается в виде прямоугольника, который дополнительно может быть разделен горизонтальными линиями на разделы или секции, как показано на рисунке 5.19. В этих разделах могут указываться имя класса, атрибуты (переменные) и операции (методы).

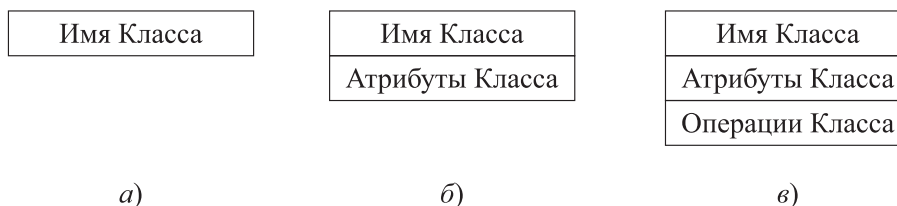


Рис. 5.19 – Графическое представление класса на диаграмме

Обязательным элементом обозначения класса является его имя. На начальных этапах разработки диаграммы отдельные классы могут обозначаться прямоугольником с указанием только имени класса (рис. 5.19, а). По мере проработки отдельных компонентов диаграммы описания классов дополняются атрибутами (рис. 5.19, б) и операциями (рис. 5.19, в).

Даже если секция атрибутов и операций является пустой, в обозначении класса она выделяется горизонтальной линией.

Имя класса

Имя класса должно быть уникальным в пределах пакета, который описывается некоторой совокупностью диаграмм классов (возможно, одной диаграммой). Оно указывается в первой верхней секции прямоугольника. В дополнение к общему правилу наименования элементов языка UML, имя класса записывается по центру секции имени жирным шрифтом и должно начинаться с заглавной буквы. Рекомендуется в качестве имен классов использовать существительные, записанные по практическим соображениям без пробелов.

Класс может не иметь экземпляров или объектов. В этом случае он называется *абстрактным* классом, а для обозначения его имени используется наклонный шрифт.



.....
В языке UML принято общее соглашение о том, что любой текст, относящийся к абстрактному элементу, записывается курсивом.
.....

В некоторых случаях необходимо явно указать, к какому пакету относится тот или иной класс. Для этой цели используется специальный символ-разделитель — двойное двоеточие «::». Синтаксис строки имени класса в этом случае имеет вид *<Имя_Пакета>::<Имя_Класса>*.

Атрибуты класса

Во второй сверху секции прямоугольника класса записываются его атрибуты или свойства. В языке UML принята определенная стандартизация записи атрибутов класса, которая подчиняется некоторым синтаксическим правилам. Каждому атрибуту класса соответствует отдельная строка текста, которая состоит из квантора видимости атрибута, имени атрибута, типа значений атрибута и, возможно, его исходного значения: *<квантор видимости><имя атрибута>:<тип атрибута> = <исходное значение>{строка-свойство}*.

Квантор видимости может принимать одно из трех возможных значений и, соответственно, отображается при помощи специальных символов:

- «+» обозначает атрибут с областью видимости типа общедоступный (*public*);
- «#» обозначает атрибут с областью видимости типа защищенный (*protected*);
- «-» обозначает атрибут с областью видимости типа закрытый (*private*).

Квантор видимости может быть опущен. В этом случае его отсутствие просто означает, что видимость атрибута не указывается. Вместо условных графических обозначений можно записывать соответствующее ключевое слово: *public*, *protected*, *private*.

Имя атрибута представляет собой строку текста, которая используется в качестве идентификатора соответствующего атрибута и поэтому должна быть уникальной в пределах данного класса. Имя атрибута является единственным обязательным элементом обозначения атрибута.

Тип атрибута представляет собой выражение, семантика которого определяется языком спецификации соответствующей модели. В нотации UML тип атрибута иногда определяется в зависимости от языка программирования, который предполагается использовать для реализации данной модели. В простейшем случае тип атрибута указывается строкой текста, имеющей осмысленное значение в пределах пакета или модели, к которым относится рассматриваемый класс.

Исходное значение служит для задания некоторого начального значения для соответствующего атрибута в момент создания отдельного экземпляра класса. Необходимо придерживаться правила принадлежности значения типу конкретного атрибута. Если исходное значение не указано, то значение соответствующего атрибута

не определено на момент создания нового экземпляра класса. С другой стороны, конструктор соответствующего объекта может переопределять исходное значение в процессе выполнения программы, если в этом возникает необходимость.

При задании атрибутов могут быть использованы две дополнительные синтаксические конструкции — это подчеркивание строки атрибута и пояснительный текст в фигурных скобках.

Строка-свойство служит для указания значений атрибута, которые не могут быть изменены в программе при работе с данным типом объектов. Фигурные скобки как раз и обозначают фиксированное значение соответствующего атрибута для класса в целом, которое должны принимать все вновь создаваемые экземпляры класса без исключения. Это значение принимается за исходное значение атрибута, которое не может быть переопределено в последующем. Отсутствие строки-свойства по умолчанию трактуется так, что значение соответствующего атрибута может быть изменено в программе.

Пример атрибута класса: «+видимость:Boolean=true» или «заработная_плата:Currency={\$500}».

Операции класса

В третьей сверху секции прямоугольника записываются операции или методы класса. Операция (*operation*) представляет собой некоторый сервис, предоставляемый экземпляром класса по определенному требованию. Каждой операции класса соответствует отдельная строка, которая состоит из квантора видимости операции, имени операции, выражения типа возвращаемого операцией значения и, возможно, строки-свойства данной операции: <квантор видимости><имя операции>(список параметров):<выражение типа возвращаемого значения>{строка-свойство}.

Квантор видимости, как и в случае атрибутов класса, может принимать одно из трех возможных значений и, соответственно, отображается при помощи специальных символов «+», «#» или «-».

Квантор видимости для операции может быть опущен. В этом случае его отсутствие просто означает, что видимость операции не указывается.

Имя операции представляет собой строку текста, которая используется в качестве идентификатора соответствующей операции и поэтому должна быть уникальной в пределах данного класса. Имя операции является единственным обязательным элементом синтаксического обозначения операции.

Список параметров является перечнем разделенных запятой формальных параметров, каждый из которых может быть представлен в следующем виде: <имя параметра>:<выражение типа>=<значение параметра по умолчанию>.

Имя параметра — это идентификатор соответствующего формального параметра. *Выражение типа* является зависимой от конкретного языка программирования спецификацией типа возвращаемого значения для соответствующего формального параметра. Наконец, *значение по умолчанию* в общем случае представляет собой выражение для значения формального параметра, синтаксис которого зависит от конкретного языка программирования и подчиняется принятым в нем ограничениям.

Выражение типа возвращаемого значения также является зависимой от языка реализации спецификацией типа или типов значений параметров, которые возвра-

щаются объектом после выполнения соответствующей операции. Двоеточие и выражение типа возвращаемого значения могут быть опущены, если операция не возвращает никакого значения. Для указания кратности возвращаемого значения данная спецификация может быть записана в виде списка отдельных выражений.

Строка-свойство служит для указания значений свойств, которые могут быть применены к данному элементу. Строка-свойство не является обязательной, она может отсутствовать, если никакие свойства не специфицированы.



Пример

В качестве примеров записи операций можно привести следующие обозначения отдельных операций:

- `+создать()` — может обозначать абстрактную операцию по созданию отдельного объекта класса, которая является общедоступной и не содержит формальных параметров. Эта операция не возвращает никакого значения после своего выполнения;
- `+нарисовать (форма:Многоугольник=прямоугольник, цвет_заливки:Color==(0, 0, 255))` — может обозначать операцию по изображению на экране монитора прямоугольной области синего цвета, если не указываются другие значения в качестве аргументов данной операции.

Отношения между классами

Кроме внутреннего устройства или структуры классов на диаграмме указываются различные отношения. При этом совокупность типов таких отношений фиксирована в языке UML и predetermined семаantikой этих типов отношений. Список отношений между классами включает в себя отношения зависимости (*dependency relationship*), ассоциации (*association relationship*), обобщения (*generalization relationship*) и реализации (*realization relationship*).

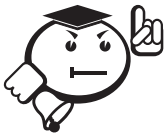
Каждое из этих отношений имеет собственное графическое представление на диаграмме, которое отражает взаимосвязи между объектами соответствующих классов. Более подробное описание этих отношений можно найти в [30].

5.5 Диаграммы последовательности

Общие сведения

Для моделирования взаимодействия объектов в языке UML используются соответствующие диаграммы взаимодействия. Диаграмма последовательности применяется для представления временных особенностей передачи и приема сообщений между объектами. Для описания структурных особенностей взаимодействия объектов используется диаграмма кооперации [29].

В соответствии с принципами объектно-ориентированного анализа информационное взаимодействие между элементами системы в рамках UML сводится к отправке и приему сообщений. Взаимодействие раскрывается в информационном аспекте коммуникации, т. е. взаимодействующие объекты обмениваются между собой некоторой информацией, которая принимает форму законченных сообщений. Однако хотя сообщение и имеет информационное содержание, оно также оказывает направленное влияние на своего получателя.



.....
 Дж. Рамбо и М. Блаха в [32] рекомендуют разрабатывать как минимум одну диаграмму последовательности для каждого прецедента. В этом случае каждая диаграмма последовательности показывает конкретную последовательность поведения варианта использования.

Рассмотрим отдельные элементы диаграммы последовательности более подробно.

Объекты

В диаграмму последовательности включаются объекты, которые непосредственно участвуют в информационном взаимодействии и не отображают возможные статические ассоциации с другими объектами. Для диаграммы последовательности ключевым моментом является именно динамика взаимодействия объектов во времени [29]. При этом диаграмма последовательности имеет два основных измерения. Одно — слева направо в виде вертикальных линий, каждая из которых изображает линию жизни отдельного объекта, участвующего во взаимодействии. Графически каждый объект изображается прямоугольником и располагается в верхней части своей линии жизни, как показано на рисунке 5.20. Внутри прямоугольника с подчеркиванием через двоеточие записываются имя объекта и имя класса, к которому данный объект принадлежит.

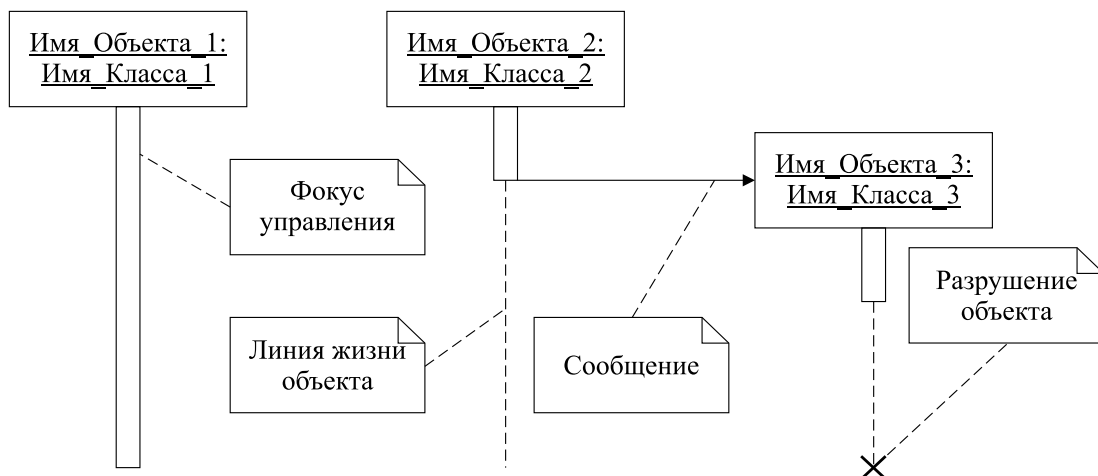


Рис. 5.20 – Графические примитивы диаграммы последовательности

Крайним слева на диаграмме изображается объект, являющийся инициатором взаимодействия. На рисунке 5.20 это Объект_1. Правее изображается другой объект, который непосредственно взаимодействует с первым. Таким образом, все объекты на диаграмме последовательности образуют некоторый порядок, определяемый степенью активности этих объектов при взаимодействии друг с другом.

Второе измерение диаграммы последовательности — вертикальная временная ось, направленная сверху вниз. Начальному моменту времени соответствует самая верхняя часть диаграммы. При этом взаимодействия объектов реализуются посредством сообщений, которые посылаются одними объектами другим. Сообщения изображаются в виде горизонтальных стрелок с именем сообщения и также образуют определенный порядок по времени своего возникновения. Другими словами, сообщения, расположенные на диаграмме последовательности выше, инициируются раньше тех, которые расположены ниже. При этом масштаб на оси времени не указывается, поскольку диаграмма последовательности моделирует лишь временную упорядоченность взаимодействий типа «раньше-позже».

Линия жизни объекта

Линия жизни объекта (*object lifeline*) изображается пунктирной вертикальной линией, ассоциированной с единственным объектом на диаграмме последовательности. Линия жизни служит для обозначения периода времени, в течение которого объект существует в системе и, следовательно, может взаимодействовать с другими объектами. Если объект существует в системе постоянно, то и его линия жизни должна продолжаться по всей плоскости диаграммы последовательности от самой верхней ее части до самой нижней, как у Объекта_2 на рисунке 5.20.

Отдельные объекты, выполнив свою роль в системе, могут быть уничтожены, чтобы освободить занимаемые ими ресурсы. Для таких объектов линия жизни обрывается в момент их уничтожения. Уничтожение объекта в рамках UML обозначается символом в форме латинской буквы «X» (Объект_3 на рисунке 5.20). Ниже этого символа пунктирная линия не изображается, поскольку соответствующего объекта в системе уже нет, и этот объект должен быть исключен из всех последующих взаимодействий [29, 30].

Фокус управления

В процессе функционирования объектно-ориентированных систем одни объекты могут находиться в активном состоянии, непосредственно выполняя определенные действия, или в состоянии пассивного ожидания сообщений от других объектов. Чтобы явно выделить подобную активность объектов, в языке UML применяется специальное понятие, получившее название фокуса управления (*focus of control*). Фокус управления изображается в форме вытянутого узкого прямоугольника, верхняя сторона которого обозначает начало получения фокуса управления объектом (начало активности), а его нижняя сторона — окончание фокуса управления (окончание активности). Этот прямоугольник располагается ниже обозначения соответствующего объекта и может заменять его линию жизни, как у Объекта_1 на рисунке 5.20, если на всем ее протяжении он является активным.

Следует отметить, что в качестве инициатора взаимодействия в системе может выступать актер или внешний пользователь. В этом случае актер изображается на диаграмме последовательности самым первым объектом слева со своим фокусом управления, как показано на рисунке 5.21.

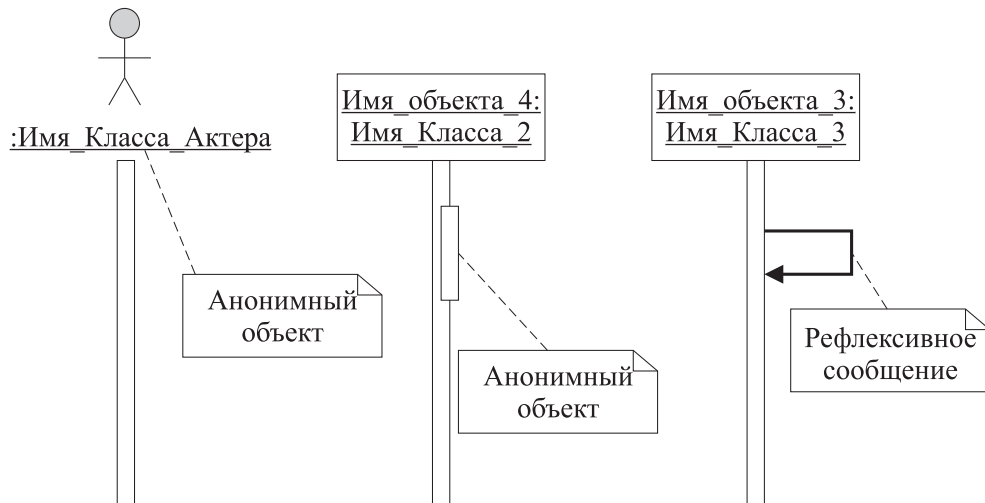


Рис. 5.21 – Фокус управления различных объектов

Наличие во многих языках программирования специальных средств построения рекурсивных процедур требует визуализации соответствующих понятий в форме графических примитивов [29]. Нотация диаграммы последовательности допускает такое рекурсивное взаимодействие объекта с самим собой. На диаграмме последовательности рекурсия обозначается небольшим прямоугольником, присоединенным к правой стороне фокуса управления того объекта, для которого изображается это рекурсивное взаимодействие.

Сообщения

Каждое взаимодействие описывается совокупностью сообщений, которыми участвующие в нем объекты обмениваются между собой. В этом смысле сообщение (*message*) представляет собой законченный фрагмент информации, который отправляется одним объектом другому. При этом прием сообщения инициирует выполнение определенных действий, направленных на решение отдельной задачи тем объектом, которому это сообщение отправлено.

Таким образом, сообщения не только передают некоторую информацию, но и требуют или предполагают от принимающего объекта выполнения ожидаемых действий. Сообщения могут инициировать выполнение операций объектом соответствующего класса, а параметры этих операций передаются вместе с сообщением.

В таком контексте каждое сообщение имеет направление от объекта, который инициирует и отправляет сообщение, к объекту, который его получает. Спецификация UML определяет несколько разновидностей сообщений, имеющих свое графическое изображение, как показано на рисунке 5.22.

Ознакомиться с различными видами сообщений подробно можно в [29–31].

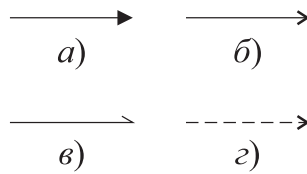


Рис. 5.22 – Различные виды сообщений между объектами на диаграмме последовательности

Обычно сообщения изображаются горизонтальными стрелками, соединяющими линии жизни или фокусы управления двух объектов на диаграмме последовательности. При этом неявно предполагается, что время передачи сообщения достаточно мало по сравнению с процессами выполнения действий объектами. За время передачи сообщения с соответствующими объектами не может произойти никаких событий: состояния объектов во время передачи сообщения остаются без изменения. Если такое предположение не может быть признано справедливым, то стрелка сообщения изображается под некоторым наклоном, так чтобы конец стрелки располагался ниже ее начала.

5.6 Диаграмма кооперации

Общие сведения

Если диаграмма последовательности служит для визуализации временных аспектов взаимодействия, то диаграмма кооперации предназначена для спецификации структурных аспектов взаимодействия. Главная особенность диаграммы кооперации заключается в возможности графически представить не только последовательность взаимодействия, но и все структурные отношения между объектами, участвующими в этом взаимодействии.

В отличие от диаграммы последовательности на диаграмме кооперации изображаются только отношения между объектами, играющими определенные роли во взаимодействии. С другой стороны, на этой диаграмме не указывается время в виде отдельного измерения. Поэтому последовательность взаимодействий и параллельных потоков может быть определена с помощью порядковых номеров. Следовательно, если необходимо явно специфицировать взаимосвязи между объектами в реальном времени, лучше это делать на диаграмме последовательности.

Поведение системы может описываться на уровне отдельных объектов, которые обмениваются между собой сообщениями, чтобы достичь нужной цели или реализовать некоторый сервис. С точки зрения аналитика или конструктора важно представить в проекте системы структурные связи отдельных объектов между собой. Такое статическое представление структуры системы как совокупности взаимодействующих объектов и обеспечивает диаграмма кооперации [29].

Кооперация

Понятие кооперации (*collaboration*) является одним из фундаментальных понятий в языке UML. Оно служит для обозначения множества взаимодействующих

с определенной целью объектов в общем контексте моделируемой системы. Цель самой кооперации состоит в том, чтобы специфицировать особенности реализации отдельных наиболее значимых операций в системе. Кооперация определяет структуру поведения системы в терминах взаимодействия участников этой кооперации.

Кооперация может быть представлена на двух уровнях. Кооперация на уровне спецификации показывает роли классификаторов и роли ассоциаций в рассматриваемом взаимодействии, а на уровне примеров — указывает экземпляры и связи, образующие отдельные роли в кооперации.

Одна и та же совокупность объектов может участвовать в различных кооперациях. При этом, в зависимости от рассматриваемой кооперации, могут изменяться как свойства отдельных объектов, так и связи между ними. Именно это отличает диаграмму кооперации от диаграммы классов, на которой должны быть указаны все свойства и ассоциации между элементами диаграммы.

Объекты

Объект представляет собой отдельный экземпляр класса и создается на этапе выполнения программы. Для графического изображения объектов используется такой же символ прямоугольника, что и для классов. Он может иметь свое собственное имя и конкретные значения атрибутов. Применительно к объектам формат строки специфицирования приобретает следующий вид: *<Имя объекта>/'<Имя роли класса>':<Имя класса>*.

Имя роли класса может не указываться. В этом случае оно исключается из строки текста вместе с последующим двоеточием. Имя роли может быть опущено в том случае, если существует только одна роль в кооперации, которую могут играть объекты, созданные на базе этого класса [29].

Таким образом, для обозначения роли достаточно указать либо имя класса (вместе с двоеточием), либо имя роли (вместе с наклонной чертой). Если роль, которую должен играть объект, наследуется от нескольких классов, то все они должны быть указаны явно и разделяться запятой и двоеточием.

Примеры изображения объектов на диаграмме кооперации приводятся на рисунке 5.23.

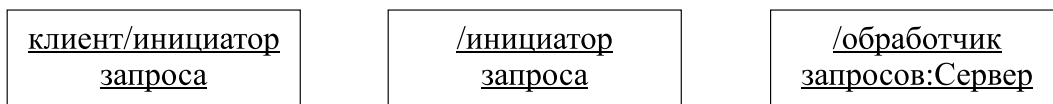


Рис. 5.23 – Примеры различных вариантов записи объектов на диаграммах кооперации

Мультиобъекты

Мультиобъект (*multiobject*) представляет собой множество объектов на одном из концов ассоциации. На диаграмме кооперации мультиобъект используется для отображения операций и сигналов, адресованных множеству объектов. Мультиобъект изображается двумя прямоугольниками, один из которых выступает из-за верхней правой вершины другого. При этом стрелка сообщения относится ко всему

множеству объектов, которые обозначают данный мультиобъект. Отношение композиции между мультиобъектом и отдельным объектом из его множества может быть указано явно, как на рисунке 5.24.

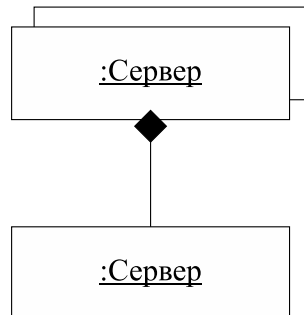


Рис. 5.24 – Изображение мультиобъектов на диаграмме кооперации

Активные объекты

В контексте языка UML все объекты делятся на две категории: пассивные и активные. Пассивный объект оперирует только данными и не может инициировать деятельность по управлению другими объектами. Однако пассивные объекты могут посылать сигналы в процессе выполнения запросов, которые они получают.

Активный объект (*active object*) имеет свою собственную нить (*thread*) управления и может инициировать деятельность по управлению другими объектами. Под нитью понимается некоторый облегченный поток управления, который может выполняться параллельно с другими вычислительными нитями или нитями управления в пределах одного вычислительного процесса или процесса управления.

Активные объекты на диаграммах обозначаются прямоугольником с более широкими границами (рис. 5.25). Каждый активный объект может инициировать единственную нить или процесс управления и представлять исходную точку потока управления. В приведенном фрагменте диаграммы кооперации активный объект «а:Вызывающий абонент» является инициатором процесса установления соединения для обмена информацией с другим абонентом (на диаграмме не показан).

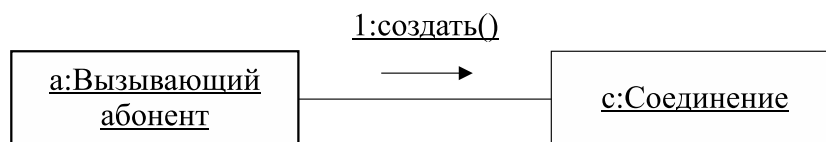


Рис. 5.25 – Графическое изображение активного объекта (слева) на диаграмме кооперации

Связи

Связь (*link*) является экземпляром ассоциации. Связь как элемент языка UML может иметь место между двумя и более объектами. Бинарная связь на диаграмме

кооперации изображается отрезком прямой линии, соединяющей два прямоугольника объектов. На каждом из концов этой линии могут быть явно указаны имена ролей данной ассоциации. Рядом с линией в ее средней части может записываться имя соответствующей ассоциации.

Связи не имеют собственных имен, поскольку полностью идентичны, будучи экземплярами ассоциации, но могут быть характеризованы с помощью *стереотипов*, которые записываются рядом с одним из ее концов и указывают на особенность реализации данной связи. В языке UML для этой цели могут использоваться следующие стереотипы [29]:

- «*association*» — ассоциация (этот стереотип используется по умолчанию, если явно не указан какой-либо другой);
- «*parameter*» — параметр метода (соответствующий объект может быть только параметром некоторого метода);
- «*local*» — локальная переменная метода (ее область видимости ограничена соседним объектом);
- «*global*» — глобальная переменная (ее область видимости распространяется на всю диаграмму кооперации);
- «*self*» — рефлексивная связь объекта с самим собой, которая допускает передачу объектом сообщения самому себе. На диаграмме кооперации рефлексивная связь изображается петлей в верхней части прямоугольника объекта.

Сообщения

Сообщения как элементы языка UML уже рассматривались ранее при изучении диаграммы последовательности. При построении диаграммы кооперации они имеют некоторые дополнительные семантические особенности. Сообщение на диаграмме кооперации специфицирует коммуникацию между двумя объектами, один из которых передает другому некоторую информацию. При этом первый объект ожидает, что после получения сообщения вторым объектом последует выполнение некоторого действия. Таким образом, именно сообщение является причиной или стимулом для начала выполнения операций, отправки сигналов, создания и уничтожения отдельных объектов. Связь обеспечивает канал для направленной передачи сообщений между объектами от объекта-источника к объекту-получателю.

Сообщения в языке UML также специфицируют роли, которые играют объекты — отправитель и получатель сообщения. Сообщения на диаграмме кооперации изображаются помеченными стрелками рядом (выше или ниже) с соответствующей связью или ролью ассоциации. Направление стрелки указывает на получателя сообщения. На диаграммах кооперации может использоваться один из четырех типов стрелок для обозначения сообщений, рассмотренных при изучении диаграммы последовательности (рис. 5.22).

На диаграмме кооперации при записи сообщений также могут использоваться стереотипы, рассмотренные ранее при построении диаграммы последовательности.

5.7 Диаграмма компонентов

Общие сведения

Диаграмма компонентов, в отличие от ранее рассмотренных диаграмм, описывает особенности физического представления системы. Диаграмма компонентов позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный, бинарный и исполняемый код. Во многих средах разработки модуль или компонент соответствует файлу. Пунктирные стрелки, соединяющие модули, показывают отношения взаимозависимости, аналогичные тем, которые имеют место при компиляции исходных текстов программ. Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними.



.....
***Компонент** — это повторно используемая часть ПО, реализующая определенное функциональное свойство системы.*
.....

Диаграмма компонентов разрабатывается для следующих целей.

1. Визуализация общей структуры исходного кода программной системы.
2. Спецификация исполнимого варианта программной системы.
3. Обеспечение многократного использования отдельных фрагментов программного кода.
4. Представление концептуальной и физической схем баз данных.

В разработке диаграмм компонентов принимают участие системные аналитики, архитекторы и программисты. Диаграмма компонентов обеспечивает согласованный переход от логического представления к конкретной реализации проекта в форме программного кода. Одни компоненты могут существовать только на этапе компиляции программного кода, другие — на этапе его исполнения. Диаграмма компонентов отражает общие зависимости между компонентами и находит широкое применение при проектировании архитектуры системы в рамках представлений компонентов и коннекторов, которые будут подробно рассмотрены в главе 6.

Компоненты

Компонент используется для представления физических сущностей. Каждый компонент реализует некоторый набор интерфейсов и служит для общего обозначения элементов физического представления модели. В качестве графического представления компонента может использоваться специальный символ — прямоугольник со вставленными слева двумя более мелкими прямоугольниками, как показано на рисунке 5.26. Внутри объемлющего прямоугольника записывается имя компонента и, возможно, некоторая дополнительная информация. Изображение этого символа может незначительно варьироваться в зависимости от характера ассоциируемой с компонентом информации.

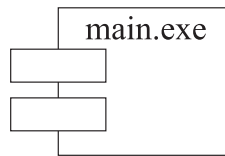


Рис. 5.26 – Графическое изображение компонента

Имя компонента подчиняется общим правилам именования элементов модели в языке UML и может состоять из любого числа букв, цифр и некоторых знаков препинания. Компонент может быть представлен на уровне типа или на уровне экземпляра. Хотя его графическое изображение в обоих случаях одинаковое, правила записи имени компонента несколько отличаются. Если компонент представляется на уровне типа, то в качестве его имени записывается только имя типа с заглавной буквы.

Если же компонент представляется на уровне экземпляра, наименование указывается в нотации <имя компонента':'имя типа>.

Поскольку конкретная реализация логического представления модели системы зависит от используемого программного инструментария, то и имена компонентов будут определяться особенностями синтаксиса соответствующего языка программирования.

Виды компонентов

Компонент как элемент физической реализации модели представляет отдельный модуль кода. Особенности реализации такого модуля могут быть проиллюстрированы с использованием дополнительных графических символов. Хотя такие дополнительные обозначения для примечаний не специфицированы в языке UML, их применение упрощает понимание диаграммы компонентов, повышая наглядность физического представления [29]. Некоторые из таких общепринятых обозначений для компонентов изображены на рисунке 5.27.

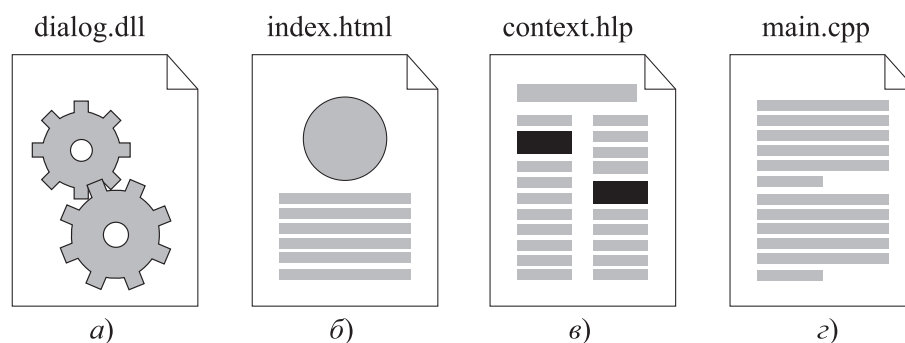


Рис. 5.27 – Варианты графического изображения компонентов

Эти элементы иногда называют артефактами, подчеркивая при этом их законченное информационное содержание, зависящее от конкретной технологии реализации соответствующих компонентов.

Интерфейсы

Следующим элементом диаграммы компонентов являются интерфейсы. В общем случае интерфейс графически изображается окружностью, которая соединяется с компонентом отрезком линии без стрелок (рис. 5.28, а). При этом имя интерфейса, которое обязательно должно начинаться с заглавной буквы «I», записывается рядом с окружностью.

Интерфейс также может быть изображен в виде прямоугольника класса со стереотипом <<интерфейс>> и возможными секциями атрибутов и операций (рис. 5.28, б). Как правило, этот вариант обозначения используется для представления внутренней структуры интерфейса, которая может быть важна для реализации.

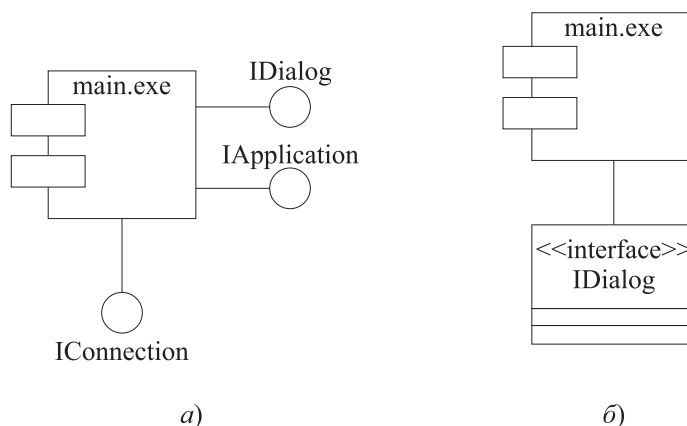


Рис. 5.28 – Графическое изображение интерфейсов на диаграмме компонентов

Зависимости

Зависимость служит для представления факта наличия такой связи, когда изменение одного элемента модели оказывает влияние или приводит к изменению другого элемента модели. Отношение зависимости на диаграмме компонентов изображается пунктирной линией со стрелкой, направленной от клиента (*зависимого элемента*) к источнику (*независимому элементу*).

Зависимости могут отражать связи модулей программы на этапе компиляции и генерации объектного кода, а также наличие в независимом компоненте описаний классов, которые используются в зависимом компоненте для создания соответствующих объектов.

В первом случае рисуют стрелку от компонента-клиента к импортируемому интерфейсу (рис. 5.29). Такая стрелка означает, что компонент не реализует соответствующий интерфейс, а использует его в процессе своего выполнения.

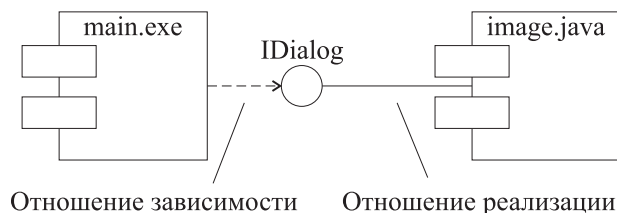


Рис. 5.29 – Фрагмент диаграммы с отношением зависимости

Другим случаем отношения зависимости на диаграмме компонентов является отношение между различными видами компонентов (рис. 5.30). Наличие подобной зависимости означает, что внесение изменений в исходные тексты программ или динамические библиотеки приводит к изменениям самого компонента. При этом характер изменений может быть отмечен дополнительно.

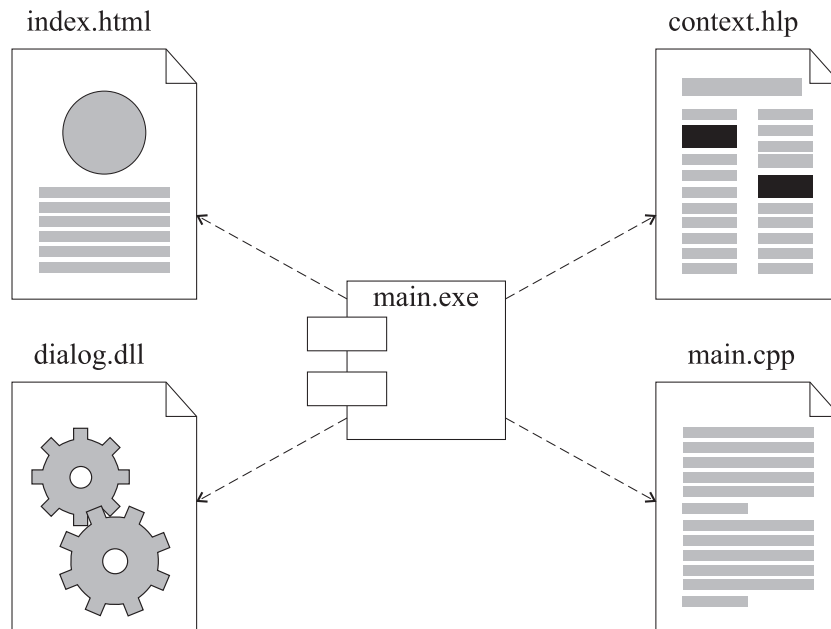


Рис. 5.30 – Графическое изображение отношения зависимости между компонентами

Наконец, на диаграмме компонентов могут быть представлены отношения зависимости между компонентами и реализованными в них классами. Эта информация имеет существенное значение для обеспечения согласования логического и физического представлений модели системы. На рисунке 5.31 приводится фрагмент зависимости подобного рода, когда некоторый компонент зависит от соответствующих классов.

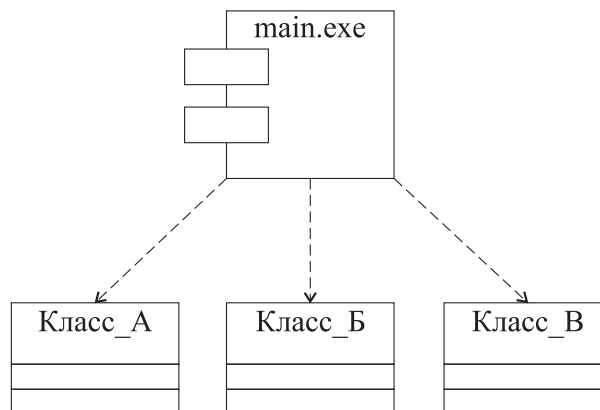


Рис. 5.31 – Графическое изображение зависимости между компонентом и классами

Объекты, которые находятся в отдельном компоненте-экземпляре, изображаются вложенными в символ данного компонента. Подобная вложенность означает, что выполнение компонента влечет выполнение соответствующих объектов. Другими словами, существование компонента в течение времени исполнения программы обеспечивает существование, а возможно, и доступ всех вложенных в него объектов. Что касается доступа к этим объектам, то он может быть дополнительно специфицирован с помощью квантификаторов видимости, подобно видимости пакетов. Содержательный смысл видимости может отличаться для различных видов пакетов.

Так, для компонентов с исходным текстом программы видимость может означать возможность внесения изменений в соответствующие тексты программ с их последующей перекомпиляцией. Для компонентов с исполняемым кодом программы видимость может характеризовать возможность запуска на исполнение соответствующего компонента или вызова реализованных в нем операций или методов.

5.8 Диаграмма развертывания

Общие сведения

Полное физическое представление программной системы включает информацию о платформах и вычислительных средствах, на которых она реализована. Такие сведения могут быть особенно важными при разработке крупных корпоративных приложений [29].

Сложные программные системы могут быть реализованы в сетевом варианте на различных вычислительных платформах и технологиях доступа к распределенным базам данных. Наличие локальной корпоративной сети требует решения целого комплекса дополнительных задач по рациональному размещению компонентов по узлам этой сети, что во многом определяет общую производительность программной системы.

Кроме того, интеграция программной системы с Интернетом определяет необходимость решения дополнительных вопросов при проектировании системы, таких как обеспечение безопасности, криптозащищенности и надежности средств доступа к информации для корпоративных клиентов. Перечисленные аспекты в немалой степени зависят от реализации проекта в форме физически существующих узлов системы, таких как серверы, рабочие станции, брандмауэры, каналы связи и хранилища данных.

Наконец, технологии доступа и манипулирования данными в рамках общей схемы «клиент-сервер» также требуют размещения больших баз данных в различных сегментах корпоративной сети, их резервного копирования, архивирования, кэширования для обеспечения необходимой производительности системы в целом.

Согласно [31] диаграммы развертывания или диаграммы размещения (*deployment diagram*) используются для того, чтобы показать распределение артефактов по узлам физической схемы создаваемой системы.

Диаграмма развертывания является диаграммой физического представления, применяется для отражения общей конфигурации и топологии распределенной информационной системы и описывает распределение компонентов по отдельным

узлам системы. Диаграмма развертывания также показывает наличие физических соединений — маршрутов передачи информации между аппаратными устройствами, задействованными в реализации системы [29].

Диаграмма развертывания предназначена для визуализации элементов и компонентов программы, существующих лишь на этапе ее исполнения (*runtime*). При этом представляются только компоненты-экземпляры программы, являющиеся исполнимыми файлами или динамическими библиотеками. Не используемые на этапе исполнения программы компоненты на диаграмме развертывания не показываются. Так, компоненты с исходными текстами программ могут присутствовать только на диаграмме компонентов.

Диаграмма развертывания содержит графические изображения процессоров, устройств, процессов и связей между ними. В отличие от диаграмм логического представления диаграмма развертывания является единой для системы в целом, поскольку должна всецело отражать особенности ее реализации. Эта диаграмма, как правило, завершает процесс ООАП для конкретной программной системы, и ее разработка, зачастую, является последним этапом спецификации модели.

Среди целей использования диаграммы развертывания выделим такие, как:

- отображение распределения компонентов системы по ее физическим узлам;
- отражение физических связей между всеми узлами реализации системы на этапе ее исполнения;
- выявление узких мест системы и реконфигурация ее топологии для достижения требуемой производительности.

Для обеспечения этих требований диаграмма развертывания разрабатывается совместно системными аналитиками, сетевыми инженерами и системотехниками.

Узлы

Узел (*node*) представляет собой некоторый физически существующий элемент системы, обладающий определенным вычислительным ресурсом. В качестве вычислительного ресурса узла может рассматриваться наличие, по меньшей мере, некоторого объема электронной или магнитооптической памяти и/или процессора. В последних версиях UML понятие узла расширено и может включать в себя не только вычислительные устройства (процессоры), но и другие механические или электронные устройства, такие как датчики, принтеры, модемы, цифровые камеры, сканеры и манипуляторы [29].

Графически на диаграмме развертывания узел изображается в форме трехмерного куба. Узел имеет собственное имя, которое указывается внутри этого графического символа. Сами узлы могут представляться как в качестве типов (рис. 5.32, а), так и в качестве экземпляров (рис. 5.32, б).

В первом случае имя узла записывается без подчеркивания и начинается с заглавной буквы. Во втором имя узла-экземпляра записывается в виде <имя узла>:'имя типа узла>. Имя типа узла указывает на некоторую разновидность узлов, присутствующих в модели системы.

Так же, как и на диаграмме компонентов, изображения узлов могут расширяться, чтобы включить некоторую дополнительную информацию о спецификации уз-

ла. Если дополнительная информация относится к имени узла, то она записывается под этим именем в форме помеченного значения (рис. 5.33).

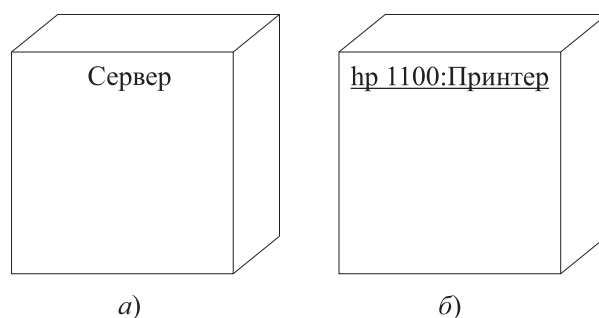


Рис. 5.32 – Графическое изображение узла на диаграмме развертывания



Рис. 5.33 – Графическое изображение узла-экземпляра с дополнительной информацией в форме помеченного значения

Если необходимо явно указать компоненты, которые размещаются на отдельном узле, то это можно сделать двумя способами [29]. Первый из них позволяет разделить графический символ узла на две секции горизонтальной линией. В верхней секции записывают имя узла, а в нижней секции — размещенные на этом узле компоненты (рис. 5.34, а). Второй способ разрешает показывать на диаграмме развертывания узлы с вложенными изображениями компонентов (рис. 5.34, б). Важно помнить, что в качестве таких вложенных компонентов могут выступать только исполняемые компоненты.

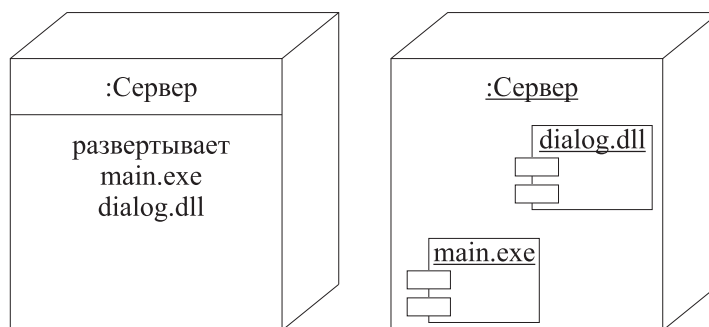


Рис. 5.34 – Варианты графического изображения узлов-экземпляров с размещаемыми на них компонентами

Соединения

Диаграмма развертывания отражает отношения между узлами. В качестве отношений выступают физические соединения между узлами и зависимости между узлами и компонентами, изображения которых тоже могут присутствовать на диаграммах развертывания.

Соединения являются разновидностью ассоциации и изображаются отрезками линий без стрелок. Наличие такой линии указывает на необходимость организации физического канала для обмена информацией между соответствующими узлами. Характер соединения может быть дополнительно специфицирован примечанием, помеченным значением, или ограничением, как показано на рисунке 5.35.

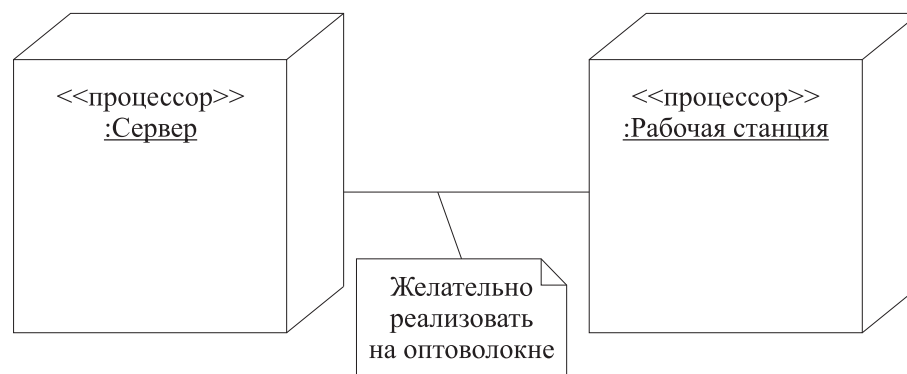


Рис. 5.35 – Фрагмент диаграммы развертывания с соединениями между узлами

Кроме соединений, на диаграмме развертывания могут присутствовать отношения зависимости между узлом и развернутыми на нем компонентами.

Подобный способ является альтернативой вложенному изображению компонентов внутри символа узла, что не всегда удобно, поскольку делает этот символ излишне объемным. Поэтому при большом количестве развернутых на узле компонентов соответствующую информацию можно представить в форме отношения зависимости (рис. 5.36).

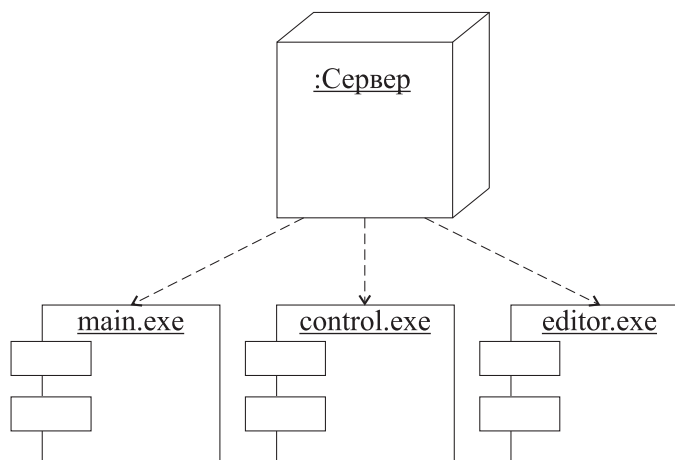


Рис. 5.36 – Диаграмма развертывания с отношением зависимости между узлом и развернутыми на нем компонентами

Диаграммы развертывания могут иметь более сложную структуру, включающую вложенные компоненты, интерфейсы и другие аппаратные устройства.

В заключение отметим, что диаграммы развертывания находят широкое применение при проектировании архитектуры программных средств в рамках представлений распределения, которые будут рассмотрены подробно в следующей главе.



Контрольные вопросы по главе 5

1. Какие модели ПО можно построить с помощью UML? Что такое модель развертывания ПО?
2. Для чего нужна диаграмма прецедентов UML?
3. Из каких основных элементов состоит диаграмма классов?
4. Какая диаграмма UML обычно используется по аналогии с блок-схемами?
5. Что такое конечный автомат? Что такое состояние в контексте диаграммы состояний UML?
6. В чем разница между состоянием действия и состоянием деятельности?
7. Какие диаграммы UML описывают физическую модель ПО?
8. Что такое узел в контексте диаграммы развертывания?
9. Для каких целей разрабатывается диаграмма компонентов? Что такое компонент?
10. Что такое линия жизни объекта в контексте диаграммы последовательности? Что такое фокус управления?

Глава 6

АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В современном мире к программному обеспечению предъявляются все более высокие требования. Пользовательские ожидания постоянно растут, и возможностей простых автономных настольных приложений становится зачастую недостаточно для удовлетворения потребностей конечных потребителей продукта [33]. Приложения все чаще взаимодействуют с другими приложениями, службами и внешними источниками данных, распространенные в прошлом монолитные архитектуры постепенно вытесняются более гибкими распределенными компонентными подходами с активным использованием сетевого взаимодействия и облачных технологий. И хотя современные инструменты и платформы разработки программных средств значительно упрощают процесс создания приложений, тщательное проектирование с учетом сценариев использования и требований заинтересованных сторон позволяет предупредить ошибки, которые могут иметь критические последствия для программного обеспечения на всех этапах его жизненного цикла.

Настоящая глава учебного пособия основана на трудах таких современных авторов-исследователей архитектуры программных средств, как Клементс П., Гарлан Д., Перри Д., Бутч Г., Басс Л., Тэйлор Р., Браун С., Бахман Ф., Норд Р., Литтл Р., Иверс Дж. и многих других современных ученых, использованные публикации которых отражены в полном объеме в списке литературы, и написана с учетом научно-практических публикаций инженеров корпораций IBM и Microsoft.

В области информационных технологий существует множество различных видов архитектуры, например помимо понятия архитектуры программного средства мы можем столкнуться с такими понятиями, как корпоративная архитектура, системная архитектура, организационная архитектура, архитектура информации, архитектура аппаратного обеспечения, архитектура приложения, архитектура инфраструктуры и так далее [2, 34, 35]. К сожалению, на сегодняшний день не существует единого общепринятого соглашения о значении каждого из этих терминов или их отношениях друг к другу, в результате чего одни и те же термины могут иметь

разные значения (омонимы), а два или более терминов зачастую могут обозначать одно и то же (синонимы) [35]. Области применения некоторых из этих терминов приведены на рисунке 6.1. И хотя в различных организациях отдельные термины могут отличаться по своему значению или даже по своему соотношению с другими терминами, на данном этапе изучения архитектуры программного обеспечения для нас важен лишь факт самого использования настоящей терминологии.

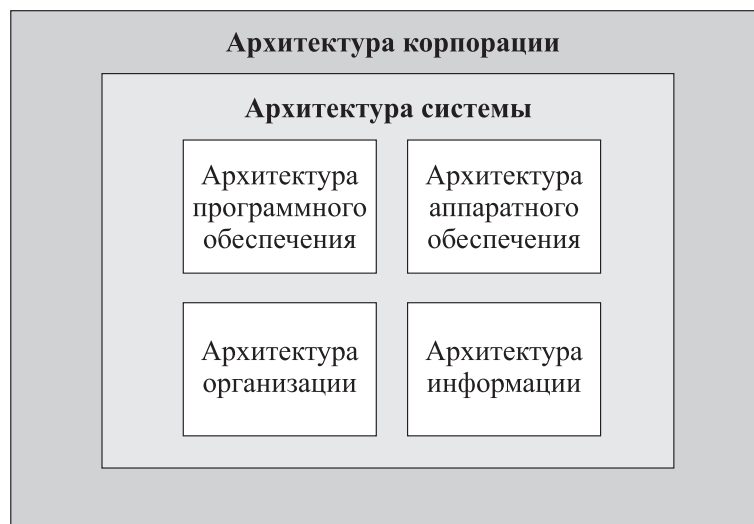


Рис. 6.1 – Области применения терминов архитектуры в рамках информационных технологий

Нижний уровень архитектуры корпорации (предприятия) представлен четырьмя элементами:

1. Архитектура программного обеспечения (*Software*), которой посвящена настоящая глава пособия.
2. Архитектура аппаратного обеспечения (*Hardware*), которая включает такие элементы, как центральное процессорное устройство (ЦПУ), память, жесткие диски, периферийные устройства, например принтер, а также элементы, используемые для их соединения.
3. Архитектура организации (*Organization*), которая включает элементы, имеющие отношение к бизнес-процессам, структурам организации, ролям и ответственности, а также основные области специализации организации.
4. Архитектура информации (*Information*), включающая структуры, которые упорядочивают информацию.

Архитектура программного обеспечения, архитектура аппаратного обеспечения, архитектура организации и архитектура информации составляют в своей совокупности целостную архитектуру системы (*System*).

Архитектура корпорации (*Enterprise*), как и архитектура системы, включает в себя программно-аппаратное обеспечение и людей. Однако архитектура корпорации сильнее связана с бизнесом, так как она концентрируется на достижении бизнес-целей и занимается такими объектами, как быстрое реагирование на возникающие ситуации и эффективность организации. Архитектура корпорации может выходить за границы компании.

Более подробное описание основных аспектов системной архитектуры и их взаимосвязей друг с другом можно найти в [34]. Основной целью данной главы настоящего пособия является рассмотрение ключевых принципов построения архитектуры программного обеспечения и существующих подходов к решению этой задачи. Однако, на наш взгляд, было бы неправильно начинать рассмотрение вопросов разработки архитектуры ПС без разъяснения понятия архитектуры ПС и целей ее построения. Разберем эти два вопроса более подробно.

6.1 Понятие архитектуры программных средств

Требования к современному программному обеспечению становятся все более сложными, поскольку пользователи ожидают все больших возможностей от используемых приложений. Возможностей простых автономных настольных приложений больше недостаточно для большинства коммерческих и деловых ситуаций [2]. В мире развитой связи приложения должны взаимодействовать с другими приложениями и службами, а также работать в различных средах, например в облаке, и на портативных устройствах. На смену распространенной в прошлом монолитной архитектуре пришло компонентное сервис-ориентированное программное обеспечение, использующее платформы, операционные системы, хост-приложения и сети для реализации функций, о которых было неизвестно всего несколько лет назад.

Эти трудности влияют не только на архитектуру, но также и на развертывание, обслуживание и администрирование программного обеспечения. Совокупная стоимость владения (*TCO*) программного обеспечения теперь в основном состоит из затрат, возникающих после развертывания [2]. Приложение с хорошо продуманной архитектурой обеспечит минимальную совокупную стоимость владения благодаря снижению затрат и времени, необходимых на развертывание приложения, обеспечение его работы, обновление для удовлетворения меняющимся требованиям и устранение проблем. Администрирование и поддержка пользователей также будут упрощены.

Сегодня архитектура программного обеспечения представляет важный раздел программной инженерии. Архитектуру, в общем, можно представить как разбиение целого на части и спецификацию взаимодействия таких систем [36]. Именно разбиение на части позволяет группам людей (отделам разработки), отдаленным друг от друга географически, находящимся в разных часовых поясах и работающим в разных организациях, эффективно сотрудничать в рамках единых проектов. Каждый такой отдел может заниматься разработкой части программного обеспечения (подсистемы), которая взаимодействует с другими компонентами ПО по детально разработанным и описанным интерфейсам, при этом имея доступ к актуальной информации, необходимой для организации такого взаимодействия. Чем сложнее программная система, тем важнее становится грамотное разбиение ее на отдельные составляющие в соответствии с запросами заинтересованных сторон, которые находят свое отражение в требованиях надежности, масштабируемости, поддерживаемости ПС и т. д.

Единая система неизбежно в процессе проектирования разбивается на составляющие по многим критериям. Каждое такое разбиение находит свое отражение

в определенной архитектуре — наборе составляющих и описаний их взаимодействия. Именно архитектура ПС является звеном, которое позволяет составным частям программной системы успешно работать вместе. Документация по архитектуре позволяет принимать обоснованные проектные решения и обеспечивать возможность поддержки ПО при кадровых изменениях в коллективе [36].

Очевидно, что разработчик архитектуры программного обеспечения должен иметь широкий диапазон как профессиональных навыков, так и навыков межличностного общения. Во время этапов анализа требования и проверки архитектор должен работать совместно с клиентом, консультироваться с партнерами и другими членами группы и выступать в качестве посредника между менеджерами, пользователями и системными администраторами. Успешные навыки межличностного общения позволяют создать лучший исходный план и более точный набор требований, сохраняя время и усилия в дальнейшем.

Создание архитектуры приложения, таким образом, представляет собой процесс формирования структурированного решения, отвечающего всем техническим и операционным требованиям и обеспечивающего оптимальные общие атрибуты качества, такие как производительность, безопасность и управляемость. Он включает принятие ряда решений на основании широкого диапазона факторов. Каждое из этих решений может иметь существенное влияние на качество, производительность, удобство обслуживания и общий успех приложения [37].

Что же такое архитектура программного средства? На сегодняшний день не существует однозначного определения. Так, на сайте института программной инженерии университета Карнеги-Меллон, занимающегося вопросами архитектуры ПС, можно найти свыше ста пятидесяти определений [38]. Однако можно постараться выделить некоторые тенденции, которые присутствуют в этих определениях, и использовать эти общие части в качестве некоей точки отсчета при разработке архитектуры. Приведем несколько наиболее интересных, с нашей точки зрения, определений.

Перри и Волф [39] предлагают по аналогии с архитектурой в строительстве следующую модель архитектуры программного обеспечения: *Архитектура = {Элементы, Форма, Отношения}*. Архитектура, согласно такому подходу, является множеством элементов определенной формы, где элементы могут быть элементами данных, вычислительными (операционными) элементами или соединительными элементами. Операционные элементы выполняют обработку элементов данных, а соединительные элементы позволяют объединить составные части архитектуры в единое целое.

Гарлан и Перри [40] рассматривают архитектуру как структуру компонент программы или системы, их взаимоотношений и принципов и руководств, управляющих изменениями такой структуры во времени.

Буч, Румбрау и Якобсон [41] представляют архитектуру как множество значимых с точки зрения организации программного обеспечения решений, набора структурных элементов, их интерфейсов, с помощью которых отдельные элементы взаимодействуют друг с другом, описаний взаимодействия таких элементов и общего архитектурного стиля, которым следует руководствоваться.

Согласно Басс, Клементсу и Казману [42] архитектура программной или вычислительной системы — это структуры системы, включающие программные эле-

менты, видимые извне свойства этих элементов и взаимоотношения между ними. Архитектура касается внешней части интерфейсов; внутренние детали элементов — детали, относящиеся исключительно к внутренней реализации, — не являются архитектурными.

Тэйлор, Медвидович и Дашофи [43] определяют архитектуру как множество принципиальных решений с точки зрения внутреннего устройства ПС, управляющих программной системой.

Согласно *IEEE Recommended Practice for Architectural Description for Software-Intensive Systems* [44] архитектура — это базовая организация системы, воплощенная в ее компонентах, их отношениях между собой и с окружением, а также принципы, определяющие проектирование и развитие системы.

Таким образом, подводя итог обзора существующих определений, можно сказать, что архитектура программного средства состоит из элементов, их отношений между собой и некоторых других аспектов, таких как множество требований или запросов заинтересованных сторон, ограничений, свойств элементов и т. д.

В рамках настоящего пособия будем придерживаться определения из [41], так как такой подход к определению архитектуры позволяет достаточно однозначно определить требования к документации на архитектуру программного средства.



.....
Архитектура программного средства (системы) — это структура системы, включающая программные элементы, видимые извне свойства этих элементов и взаимоотношения между ними.

Для более точного описания понятия архитектуры введем несколько дополнительных связанных с ним определений [44].



.....
Система — это набор компонентов, объединенных для выполнения определенной функции или набора функций.

Термин «система» охватывает отдельные приложения, системы в традиционном смысле, подсистемы, системы систем, линейки продуктов, семейства продуктов, целые корпорации и другие агрегации, имеющие отношение к данной теме. Система существует для выполнения одной или более миссий в своем окружении.

Окружение, или контекст, определяет ход и обстоятельства экономических, эксплуатационных, политических и других влияний на систему.



.....
Миссия — это применение или действие, для которого одно или несколько заинтересованных лиц планируют использовать систему в соответствии с некоторым набором условий.

Заинтересованное лицо — это физическое лицо, группа или организация (или ее категории), которые заинтересованы в системе или имеют связанные с ней задачи.

Также следует отметить частое использование в определениях термина «компонент». Тем не менее большая часть определений архитектуры не определяет значение этого термина, чтобы оно соответствовало множеству толкований, возможных в отрасли [35]. Компонент может быть логическим или физическим, технологически-независимым или технологически-связанным.

Обозначив границы термина «архитектура программного обеспечения», перейдем к рассмотрению необходимости ее использования на практике.

6.2 Целесообразность разработки и документирования архитектуры

Тщательная разработка и документирование архитектуры требуют значительных затрат временных и материальных ресурсов и, зачастую, могут рассматриваться разработчиками и менеджерами как пустая трата ресурсов. Разберем вопрос целесообразности вложений в документирование архитектуры как основы программного средства или системы. Для обоснования такой целесообразности определим основные цели, которые решаются при проектировании архитектуры [37].

Архитектура приложения должна объединять бизнес-требования и технические требования через понимание вариантов использования с последующим нахождением путей их реализации в ПО. Цель архитектуры — выявить требования, оказывающие влияние на структуру приложения. Хорошая архитектура снижает бизнес-риски, связанные с созданием технического решения. Хорошая структура обладает значительной гибкостью, чтобы справляться с естественным развитием технологий как в области оборудования и ПО, так и пользовательских сценариев и требований. Архитектор должен учитывать общий эффект от принимаемых проектных решений, обязательно присутствующие компромиссы между атрибутами качества (такими как производительность и безопасность) и компромиссы, необходимые для выполнения пользовательских, системных и бизнес-требований.

Следовательно, архитектура приложения должна включать в себя решения по следующим вопросам [2]:

- обработка исключений и ведение журналов;
- производительность, масштабируемость, доступность и другие показатели качества;
- соответствие нормативным документам и требованиям;
- реальные ограничения окружающей среды;
- соответствие требованиям поддержки и обслуживания программного обеспечения;
- согласованность структуры программного средства и подходов к решению функциональных задач в рамках всего проекта;
- возможность оценки эффективности архитектуры.

Таким образом, архитектура представляет собой базовое руководство по разработке программного продукта, обеспечивает лучшую координацию между отдельными частями проекта, позволяет заложить фундамент для реализации нефункци-

ональных требований к системе и ввести единые стандарты разработки и взаимодействия модулей конечного продукта, а также вести учет и управление рисками проекта. Необходимо также помнить, что архитектура должна раскрывать структуру системы, но скрывать детали реализации; реализовывать все варианты использования и сценарии; по возможности отвечать всем требованиям различных заинтересованных сторон; выполнять требования как по функциональности, так и по качеству [37].



.....
Неправильно выработанная архитектура обуславливает нестабильность ПО, невозможность поддерживать существующие или будущие бизнес-требования, сложности при развертывании или управлении в среде производственной эксплуатации.
.....

Кроме того, необходимо отметить, что даже лучшая архитектура, идеально подходящая для конкретной программной системы, становится практически бесполезной в том случае, когда люди, которые должны ее использовать, не понимают, как она работает и как ее использовать на практике. Недостаточно только разработать архитектуру, после этого необходимо ее грамотно описать таким образом, чтобы все заинтересованные стороны имели доступ к ее структурированному описанию и могли найти всю необходимую информацию по проекту.



.....
Фактически, именно документация выступает за системного архитектора. Она говорит за него сегодня, когда архитектор занят другими проектами, и, что еще более важно, документация будет говорить за архитектора завтра, когда он покинет проект и на его место придет новый человек [2].
.....

Хотя документирование зачастую рассматривается как необходимая процедура, которую архитектору приходится выполнять по условиям контракта или требованиям клиента, наиболее опытные архитекторы разрабатывают документацию не потому, что «так надо», а потому что они видят всю значимость хорошей документации для создания качественного конечного продукта [36].

Сегодня документация на архитектуру служит различным целям. Выделим среди них основные три [36].

1. *Обучение.*

Документация по архитектуре программного средства используется при включении в проект новых людей (среди которых могут быть новые члены команды разработчиков, эксперты-аналитики или даже новый системный архитектор).

2. *Общение между заинтересованными сторонами.*

Такое общение необходимо для формирования и реализации как функциональных, так и нефункциональных требований к продукту. Основные заинтересованные стороны и их описание представлены в таблице 6.1.

Таблица 6.1 – Заинтересованные стороны в архитектуре программного средства

Заинтересованная сторона	Роль в рамках жизненного цикла ПО	Как использует документацию по архитектуре ПО
Системный архитектор	Ответственный за разработку архитектуры и сопутствующей документации	Разработка документации. Описание архитектурных решений и доказательств их эффективности
Бизнес-аналитик	Ответственный за описание модели предметной области и бизнес-процессов	Оценка соответствия архитектуры бизнес-целям
Потребитель программного продукта	Выступает заказчиком разработки, зачастую представляет конечного пользователя продукта	Оценка соответствия конечного продукта функциональным требованиям, требованиям качества программного обеспечения, сроков и стоимости реализации
Администратор СУБД	Принимает участие в проектировании хранилища данных	Получение сведений об использовании данных и структур данных в конечном продукте
Специалист по внедрению системы	Ответственный за установку готового программного продукта	Получение сведений о компонентах архитектуры, которые должны быть размещены на рабочих станциях и серверах заказчика и их роли в структуре программного продукта
Разработчик программного обеспечения	Ответственный за проектирование программного продукта на основе документации по архитектуре	Получение сведений о взаимодействии компонент архитектуры между собой и общих требований к их организации
Служба технической поддержки	Отвечает за поддержку программного продукта в течение всего жизненного цикла	Получение сведений о последствиях внесения изменений в программный продукт
продолжение на следующей странице		

Таблица 6.1 – Продолжение

Заинтересованная сторона	Роль в рамках жизненного цикла ПО	Как использует документацию по архитектуре ПО
Системный администратор	Ответственный за поддержку аппаратного и программного обеспечения в компьютерной сети. Принимает участие во внедрении и настройке системы	Получение сведений о нагрузках на аппаратное обеспечение и требованиях к нему в различных режимах использования программного продукта
Менеджер проекта	Ответственный за планирование разработки и выделение необходимых ресурсов	Получение сведений, необходимых для формирования календарного плана и бюджета проекта
Инженер качества программного продукта	Ответственный за тестирование конечного программного продукта на соответствие формальным требованиям качества	Разработка методики тестирования конечного программного продукта
Пользователь программного продукта	Конечный пользователь программного средства	Получение сведений о реализации конкретных функциональных возможностей и общем устройстве программного средства

3. Руководство по проектированию внутренней структуры продукта и его анализу.

Именно документация по архитектуре программного средства отвечает на вопросы о том, что проектировать и разрабатывать и как происходит учет нефункциональных требований в системе.

Следует отметить, что, представляя собой сложную многоуровневую структуру, архитектура приложения не может быть описана какой-либо одной диаграммой, так как любая такая попытка приведет к одностороннему описанию программного средства. Вместо этого необходимо разрабатывать отдельные *архитектурные представления* с точки зрения каждого важного архитектурного аспекта.

При таком подходе различные архитектурные представления по-разному раскрывают отдельные цели и задачи, стоящие при проектировании системы. Так, например, послойное архитектурное представление определяет набор решений в рамках вопросов переносимости программного средства, а представление развертывания ПО может определять надежность и стабильность программного средства. Таким образом ответ на вопрос, какие представления использовать при описании

архитектуры, полностью зависит от задач, которые тот или иной набор документации по архитектуре должен решать.



.....
 Авторы считают необходимым отдельно отметить отсутствие единого и универсального списка представлений, которые обязательно должны быть использованы в строго определенном порядке. Важность того или иного представления определяется исключительно целями, стоящими перед архитектором.

В соответствии с такой концепцией представлений даже архитектура одного приложения может быть описана разными архитекторами различными способами.

Несмотря на все многообразие возможных архитектурных представлений, каждое такое представление состоит из определенных частей.

1. *Графическое описание*, которое отражает основные элементы архитектурного представления и их взаимодействие между собой.
2. *Каталог элементов представления*, который содержит список всех элементов данного представления и их характеристик.
3. *Описание поведения элементов* и интерфейсов их взаимодействия.
4. *Характеристика изменчивости элементов* представления и возможности их адаптации в условиях меняющихся требований.
5. Обоснование использованных методик, подходов и шаблонов проектирования.

В дополнение к описанию отдельных представлений документация по архитектуре программного средства также, как правило, включает в себя положения, общие для всех представлений. К таковым относятся:

1. Введение, которое содержит сведения, необходимые для эффективного использования документации.
2. Описание соотношения отдельных архитектурных представлений друг с другом.
3. Обоснование основных концепций, использованных при построении всей архитектуры.

Хотя каждый из описанных пунктов документации может быть представлен разными способами, все подходы к документированию архитектуры программных средств можно объединить в три большие группы [36].

1. Неформальное описание.

Данный подход к документированию характеризуется наличием графического представления архитектуры ПО и его семантическим описанием на естественном языке.

2. Полуформальное описание.

Данный подход предполагает описание архитектурных представлений с помощью стандартизированных нотаций, таких как UML, при этом не предъявляя строгих требований к семантическому значению отдельных элементов представлений.

3. Формальное описание.

Для формального описания характерно использование однозначной (как правило, математически описанной) семантики, благодаря чему становится возможным формальный анализ как синтаксиса описания, так и его семантики. Существует множество формальных нотаций, однако ни одна из них не имеет достаточно широкого распространения. Такие нотации получили название языков описания архитектуры (ADL). Основное их преимущество заключается в возможностях автоматизации задач, связанных с реализацией разработанной архитектуры, и использования средств генерации кода.

Большинство примеров в рамках данного пособия приведены в полужформальной нотации.

6.3 Проектирование архитектуры программных средств

Основные принципы проектирования архитектуры включают в себя следующие утверждения [37].

- Создавайте, чтобы изменять. Продумайте, как со временем может понадобиться изменить приложение, чтобы оно отвечало вновь возникающим требованиям и задачам, и предусмотрите необходимую гибкость.
- Создавайте модели для анализа и сокращения рисков. Используйте средства проектирования, системы моделирования, такие как унифицированный язык моделирования, и средства визуализации, когда необходимо выявить требования, принять архитектурные и проектные решения и проанализировать их последствия. Тем не менее не создавайте слишком формализованную модель, она может ограничить возможности для выполнения итераций и адаптации дизайна.
- Используйте модели и визуализации как средства общения при совместной работе. Для построения хорошей архитектуры критически важен эффективный обмен информацией о дизайне, принимаемых решениях и вносимых изменениях. Используйте модели, представления и другие способы визуализации архитектуры для эффективного обмена информацией и связи со всеми заинтересованными сторонами, а также для обеспечения быстрого оповещения об изменениях в дизайне.
- Выявляйте ключевые инженерные решения. В самом начале проекта уделите достаточное количество времени и внимания для принятия правильных решений, это обеспечит создание более гибкого дизайна, внесение изменений в который не потребует полной его переработки.

Приступая к работе над архитектурой приложения, необходимо также помнить об основных принципах проектирования в целом. Это поможет создать архитектуру, которая будет следовать проверенным подходам, обеспечит минимизацию затрат, простоту обслуживания, удобство использования и расширяемость. Среди наиболее распространенных принципов отметим следующие.

- Разделение функций. Разделите приложение на отдельные компоненты, по возможности, с минимальным перекрытием функциональности. Важным

фактором является предельное уменьшение количества точек соприкосновения, что обеспечит высокую связность (*high cohesion*) и слабую связанность (*low coupling*). Неверное разграничение функциональности может привести к высокой связанности и сложностям взаимодействия, даже несмотря на слабое перекрытие функциональности отдельных компонентов.

- Принцип единственности ответственности. Каждый отдельно взятый компонент или модуль должен отвечать только за одно конкретное свойство/функцию или совокупность связанных функций.
- Принцип минимального знания (также известный как Закон Деметера (*Law of Demeter, LoD*)). Компоненту или объекту не должны быть известны внутренние детали других компонентов или объектов.
- Не повторяйтесь (*Don't repeat yourself, DRY*). Намерение должно быть обозначено только один раз. В применении к проектированию приложения это означает, что определенная функциональность должна быть реализована только в одном компоненте и не должна дублироваться ни в одном другом компоненте.
- Минимизируйте проектирование наперед. Проектируйте только то, что необходимо. В некоторых случаях, когда стоимость разработки или издержки в случае неудачного дизайна очень высоки, может потребоваться полное предварительное проектирование и тестирование. В других случаях, особенно при гибкой разработке, можно избежать масштабного проектирования наперед (*big design upfront, BDUF*). Если требования к приложению четко не определены или существует вероятность изменения дизайна со временем, старайтесь не тратить много сил на проектирование раньше времени. Этот принцип называют YAGNI («*You ain't gonna need it*»).

6.4 Понятие архитектурного стиля

Как правило, архитектура программного средства редко создается без опоры на уже существующие решения в рамках предметной области. В качестве основы для принятия решений при проектировании архитектуры обычно используется определенный архитектурный стиль.



.....
Архитектурный стиль, иногда называемый архитектурным шаблоном или парадигмой, — это набор принципов, высокоуровневая схема, обеспечивающая абстрактную инфраструктуру для семейства систем [36].

Архитектурный стиль улучшает секционирование и способствует повторному использованию дизайна благодаря обеспечению решений часто встречающихся проблем. Архитектурные стили и шаблоны можно рассматривать как набор принципов, формирующих приложение.

В настоящем пособии приводятся обобщенные шаблоны и принципы, применяемые при проектировании приложений сегодня (табл. 6.2). К ним относятся такие шаблоны, как клиент-сервер, многоуровневая архитектура, компонентная архитектура, архитектура, основанная на шине сообщений, и сервисно-ориентированная архитектура (*service-oriented architecture, SOA*).

Таблица 6.2 – Краткое описание основных архитектурных стилей

Архитектурный стиль/парадигма	Ключевые особенности стиля
Клиент-сервер (тип представления компонентов и коннекторов)	Система разделяется на два приложения, где клиент выполняет запросы к серверу. Во многих случаях в роли сервера выступает база данных, а логика приложения представлена процедурами хранения
Компонентная архитектура (тип модульного представления)	Дизайн приложения разлагается на функциональные или логические компоненты с возможностью повторного использования, предоставляющие тщательно проработанные интерфейсы связи
Дизайн на основе предметной области (тип модульного представления)	Объектно-ориентированный архитектурный стиль, ориентированный на моделирование сферы деловой активности и определяющий бизнес-объекты на основании сущностей этой сферы
Многослойная архитектура (тип модульного представления)	Функциональные области приложения разделяются на многослойные группы (уровни)
Шина сообщений (тип представления компонентов и коннекторов)	Архитектурный стиль, предписывающий использование программной системы, которая может принимать и отправлять сообщения по одному или более каналам связи, так что приложения получают возможность взаимодействовать, не располагая конкретными сведениями друг о друге
N-уровневая/3-уровневая архитектура (тип представления компонентов и коннекторов)	Функциональность выделяется в отдельные сегменты, во многом аналогично многослойному стилю, но в данном случае сегменты физически располагаются на разных компьютерах
Объектно-ориентированная архитектура (тип модульного представления)	Парадигма проектирования, основанная на распределении ответственности приложения или системы между отдельными многократно используемыми и самостоятельными объектами, содержащими данные и поведение
продолжение на следующей странице	

Таблица 6.2 – Продолжение

Архитектурный стиль/парадигма	Ключевые особенности стиля
Сервисно-ориентированная архитектура (СОА) (тип представления компонентов и коннекторов)	Описывает приложения, предоставляющие и потребляющие функциональность в виде сервисов с помощью контрактов и сообщений



.....

Важно понимать, что стили описывают разные аспекты приложений. Некоторые архитектурные стили описывают схемы развертывания, некоторые — вопросы структуры и дизайна, другие — аспекты связи.

.....

Таким образом, типовое приложение, как правило, использует сочетание нескольких рассматриваемых в данной главе стилей.

В соответствии с используемым нами определением архитектуры ПО для построения архитектуры необходимо ответить на три вопроса.

1. Из каких структурных модулей состоит программное обеспечение?
2. Из каких исполняемых элементов, характеризующихся определенным взаимодействием, состоит программное обеспечение?
3. Как данное программное обеспечение связано с непрограммными структурами и окружением?

Каждому из поставленных вопросов соответствует один из трех типов архитектурных стилей:

- 1) тип модульного представления;
- 2) тип представления компонентов и коннекторов;
- 3) тип представления распределения.

Тип модульного представления

Стили архитектуры, составляющие тип «модульного» представления (*«module» viewtype*), используются для определения декомпозиции и организации функциональных возможностей и для реализации в виде согласованного комплекса программного кода. Эти стили используются для описания структуры программных блоков, для определения отдельных блоков, для документирования функциональных ответственностей каждого блока и для определения характеристик взаимосвязей между блоками. Использование наборов типов модульного представления, отражающих декомпозицию функциональных возможностей — от высокоуровневой абстракции до уровня подробностей реализации, — позволяет обмениваться информацией о прослеживаемости требований и обеспечивать основу для анализа воздействия.

Способ разбиения программного обеспечения на отдельные модули является одной из наиболее важных форм описания его структуры. Такое разбиение не только определяет, каким образом исходный код разбивается на отдельные файлы, но и описывает знания отдельных модулей о сервисах, предоставляемых другими модулями, и содержит сведения о том, как отдельные модули объединяются в более крупные подсистемы, а также о глобальных структурах данных, оказывающих влияние на несколько модулей одновременно или на всю программную систему в целом [36].

Кроме того, именно модульное представление системы зачастую позволяет отслеживать влияние, оказываемое в результате изменения отдельных компонентов программного обеспечения на другие составляющие ПО.

Элементами модульного представления являются модули.



.....
***Модуль** — это единица реализации программного обеспечения, предоставляющая определенное множество функциональных возможностей.*

Документирование последовательностей вызова модулей позволяет обнаружить перекрытие функциональных возможностей и помогает в анализе недочетов. Блоки связаны между собой интерфейсами. Можно по желанию либо использовать некоторые данные интерфейсов из предыдущих функциональных блоков, либо использовать данные только из текущего блока, инкапсулировав (скрыв) информацию из предшествующей обработки. Элементы в типах модульных представлений обычно сопоставляются с элементами в стилях архитектуры, которые находятся в типах представлений компонентов и коннекторов. Очень важно не перегружать модульные представления элементами из других типов представлений. В случае, если сочетаются несколько стилей архитектуры, следует поддерживать разделение информации в соответствующих представлениях [2].

В стиле архитектуры «декомпозиция» (*decomposition*) между модульными элементами используется отношение по принципу «один является частью другого» (*is-part-of*). Обычно в нем отражается декомпозиция блока на элементы блока или разбивка функциональных ответственностей по модулям. В стиле «использует» (*uses*) указывается зависимость между модульными элементами в отношении по принципу «один использует другой». В нем документируется, какие модули необходимы для корректной работы модуля, который их использует. В стиле «обобщение» (*generalization*) отражаются отношения между модульными элементами по принципу «один является другим». Родительский (или обобщающий) элемент документирует ответственности, общие для всех дочерних (специализированных) элементов. В «многослойном» (*layered*) стиле отражается стек виртуальных машин с отношением «одному разрешено использовать другой» (*allowed-to-use*). Многослойная архитектура обеспечивает высокий уровень модифицируемости и переносимости. Наконец, стиль «модель данных» (*data model*) используется для отображения отношений между сущностями данных.

Описание каждого модуля может включать в себя следующие характеристики:

- *Наименование* является одной из основных характеристик модуля и, как правило, соответствует роли модуля в системе, а также может указывать

на иерархическое положение модуля в программной системе (например, название *А.Б.В* может указывать на то, что модуль *В* является подмодулем *Б*, который в свою очередь является подмодулем *А*).

- *Область ответственности модуля*, которая отражает роль модуля в системе и уточняет круг функциональных возможностей модуля, приближенное представление о котором можно получить из наименования модуля.
- *Видимость интерфейсов*. В случае, если модуль имеет дочерние элементы, каждый из дочерних элементов может иметь свои интерфейсы, доступные или недоступные из-за пределов родительского модуля. На рисунке 6.2 показаны два основных способа реализации интерфейсов дочерних модулей.

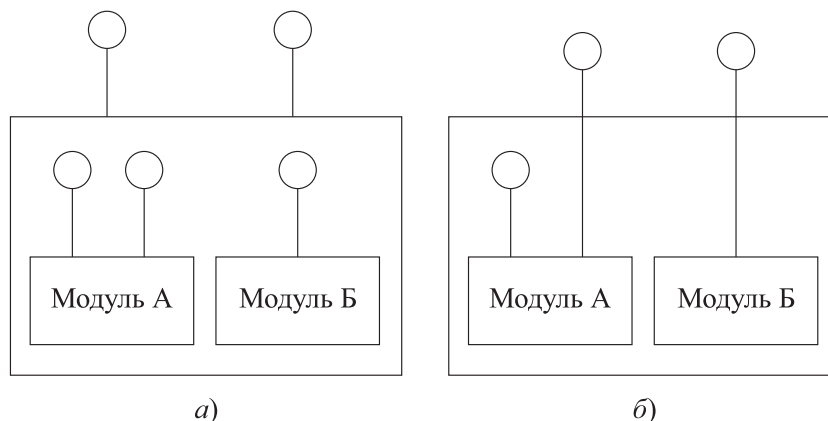


Рис. 6.2 – Реализация интерфейсов дочерних модулей

В случае «а» родительский модуль имеет собственный интерфейс для взаимодействия со внешними модулями, в то время как интерфейсы дочерних модулей используются только внутри родительского. В случае «б» множество интерфейсов родительского модуля является подмножеством множества интерфейсов дочерних модулей.

- *Сведения для реализации*. Хотя сведения для реализации не являются частью архитектуры в чистом виде, они могут быть полезны при проектировании и реализации модуля. Такие сведения могут включать в себя описание файлов с исходным кодом, информацию, необходимую для тестирования модуля (например, список тестовых ситуаций), информацию, которая может потребоваться менеджеру проекта (например, сведения о бюджете и сроках реализации), дополнительные требования и ограничения, накладываемые на модуль, которые важны с точки зрения архитектуры, но не нашли своего отражения в рамках интерфейсов и описания модуля.

Архитектурные стили типа модульного представления используются в качестве базового плана по разработке программного продукта. Кроме того, модульное представление позволяет выполнять трассировку и отслеживание требований заинтересованных сторон. Наконец, именно модульное представление программного продукта удобно использовать для объяснения его архитектуры и структуры новым членам команды или другим заинтересованным лицам, незнакомым с системой.

Рассмотрим основные типы модульных представлений архитектуры программных средств более подробно.

Стиль «декомпозиция»

Представление декомпозиции описывает организацию кода в рамках модулей и подмодулей и позволяет выделить зоны ответственности модулей и, таким образом, разделить функциональные возможности. Большинство специалистов начинают построение архитектуры именно с этого стиля.

Для разбиения модулей на составляющие существует ряд критериев.

- Достижение определенных показателей качества. Например, для реализации модифицируемости программного кода все части программного обеспечения, наиболее подверженные изменениям во времени, можно инкапсулировать в отдельные модули, тем самым минифицировав потребность внесения изменений в других модулях в будущем.
- Разработка или приобретение. Некоторые модули могут быть приобретены на коммерческих условиях или быть доступными в рамках открытого и свободного программного обеспечения. Такие модули реализуют часть функциональных требований, а оставшиеся функциональные возможности должны быть реализованы смежными подсистемами, что необходимо учитывать при проектировании архитектуры программного обеспечения в целом.
- Выделение общих модулей в рамках семейства программных продуктов. Эффективная реализация нескольких смежных программных продуктов в рамках одного семейства требует выделения общей функциональности в отдельные модули, используемые большинством продуктов.
- Параллельное выполнение работ. Различные функциональные требования могут реализовываться параллельно, для чего соответствующие функциональные возможности необходимо выделять в отдельные модули, работа над которыми ведется разными командами разработчиков. При этом список работ, которые могут быть выполнены параллельно друг другу, зависит в том числе от состава команды разработчиков. Так, например, при наличии сильных веб-разработчиков, логичным шагом является выделение в отдельные модули веб-компонент приложения.

Элементами представления декомпозиции являются модули, которые зачастую группируются в подсистемы (рис. 6.3). Основным отношением в рамках данного стиля является отношение обобщения.

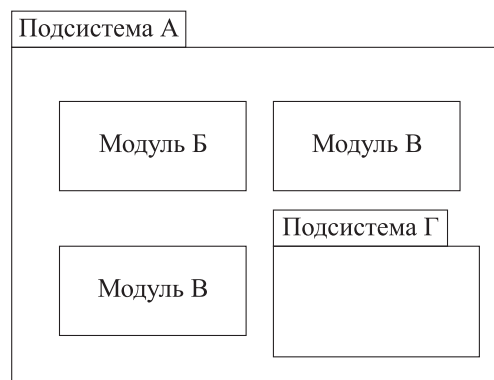


Рис. 6.3 – Архитектурный стиль «декомпозиция»

Стиль «использует»

Представления стиля «использует» основаны на отношении использования, которое является частным случаем отношения зависимости. Будем считать, что модуль А программы использует другой модуль Б, если правильное функционирование А зависит от правильности функционирования Б (рис. 6.4).

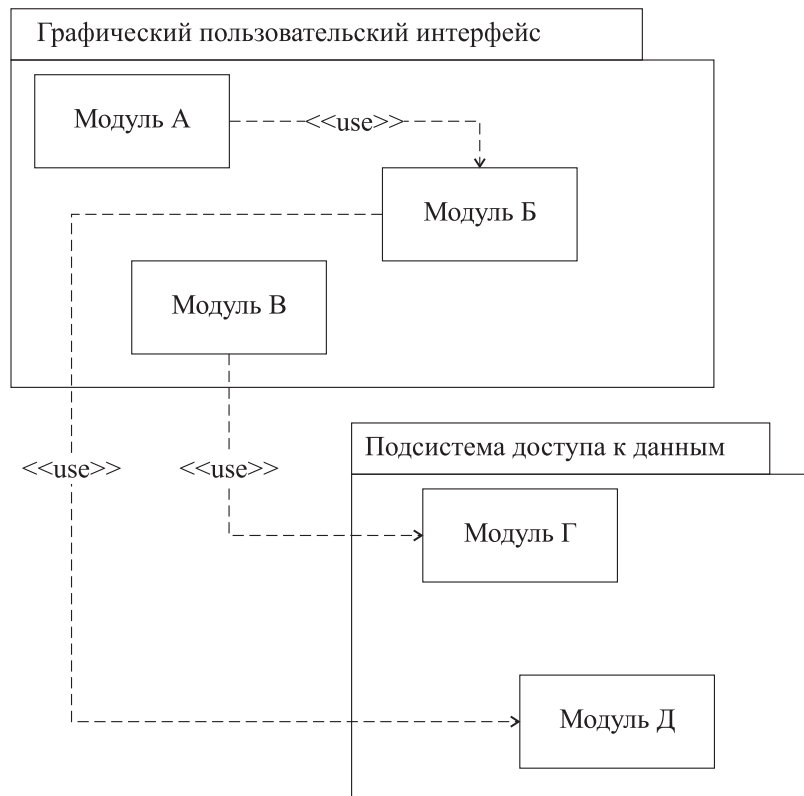


Рис. 6.4 – Архитектурный стиль «использует»

В то время как стиль «декомпозиция» показывает только разделение программного обеспечения на модули и подсистемы, стиль «использует» отражает существующие зависимости между составными частями программного обеспечения и содержит сведения о том, какие модули системы должны быть реализованы для выполнения определенной части функциональных требований. Использование данного стиля позволяет реализовать инкрементные разработку и внедрение программного продукта. Кроме того, стиль «использует» позволяет планировать процессы тестирования, отладки и внесения изменений в программное обеспечение.

На рисунке 6.5 представлено первичное представление архитектурного стиля «использует» с явным указанием интерфейсов модулей.

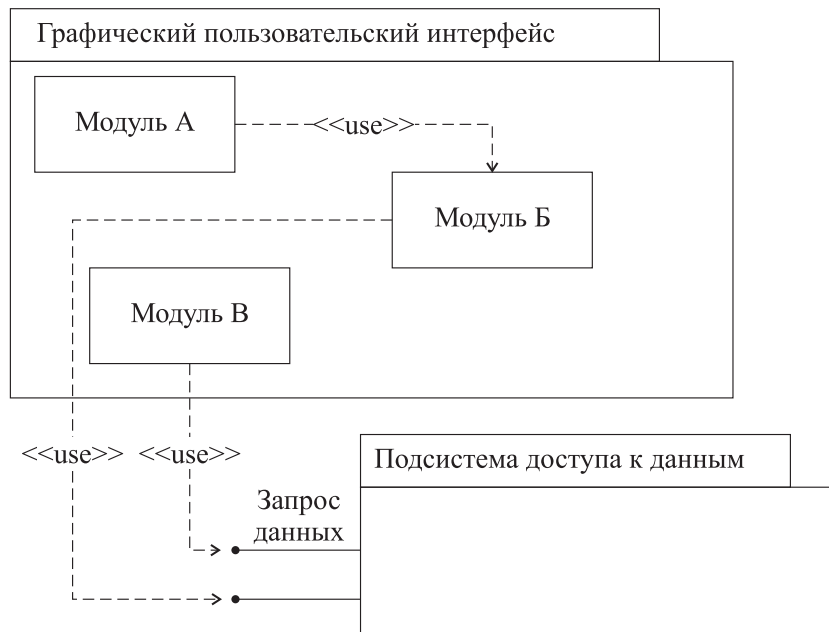


Рис. 6.5 – Архитектурный стиль «использует» с явным указанием интерфейсов

Стиль «обобщение»

Стиль «обобщение» используется в случае, если необходимо предусмотреть расширяемость и возможности эволюции программного продукта. Разбиение на модули производится на том основании, что отдельные модули имеют некоторые сходства и вариации. Родительский элемент отношения обобщения является более общей версией дочернего модуля, и наоборот, дочерний модуль является более детальной версией родительского модуля (таким образом, вся общая функциональность может быть собрана в рамках родительского модуля, а изменяющаяся часть функциональности реализуется дочерним модулем). Пример отношения обобщения приведен на рисунке 6.6.

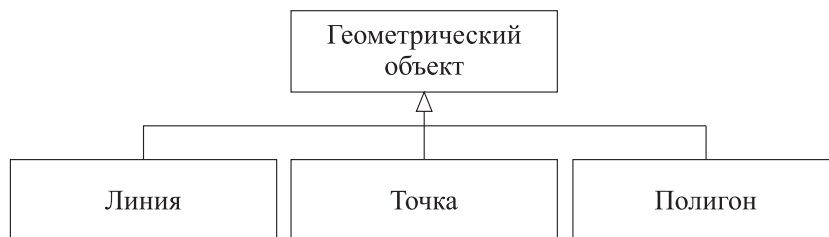


Рис. 6.6 – Отношение обобщения

«Многослойный» стиль

Элементами «многослойного» стиля являются слои. Каждый слой представляет собой набор модулей, которые посредством множества публичных интерфейсов предоставляют определенный набор услуг. Отношения между слоями являются однонаправленными отношениями «одному разрешено использовать другой». При

этом направление таких отношений строго определено — от более высокоуровневого слоя к менее высокоуровневому слою. Таким образом, схема, приведенная на рисунке 6.7, не соответствует «многослойному» стилю, несмотря на то, что программное обеспечение разделено на «слои», так как слой более низкого уровня (Слой В) может использовать интерфейсы более высокоуровневого слоя (Слой Б). Правильное использование данного стиля показано на рисунках 6.8 и 6.9. В случае на рисунке 6.8 подразумевается, что каждый слой может использовать только сервисы, предоставляемые модулями следующего слоя, на рисунке 6.9 возможные взаимодействия между модулями указаны явно.

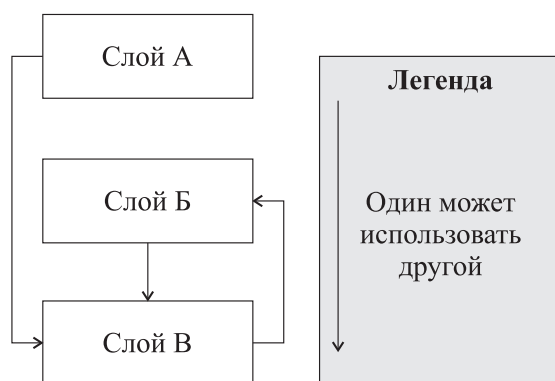


Рис. 6.7 – Неправильное использование «многослойного» стиля архитектуры

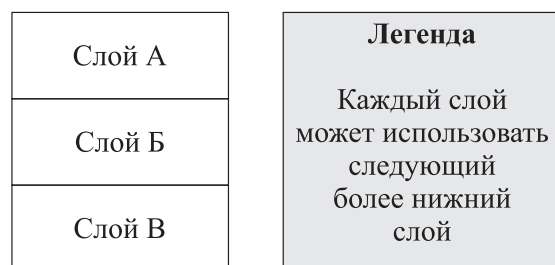


Рис. 6.8 – Представление «многослойного» стиля, в соответствии с которым модули каждого уровня могут использовать сервисы модулей следующего уровня

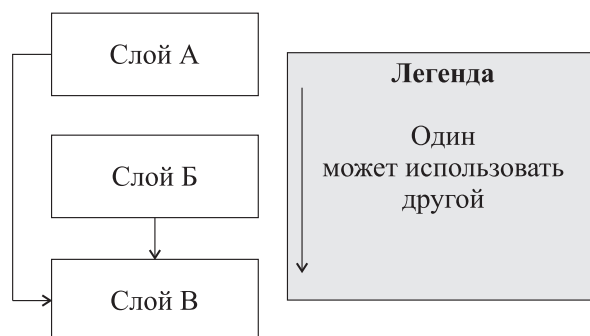


Рис. 6.9 – «Многослойный» архитектурный стиль с явно указанными возможными вариантами взаимодействия между слоями

Слой может включать в себя любые модули, например веб-компоненты или модули взаимодействия с СУБД. Единственное ограничение, накладываемое на элементы слоя, — наличие интерфейса, который может быть использован другим модулем.

Содержимое слоя может включать как весь список модулей слоя целиком, так и лишь указания о том, какие модули должны быть реализованы в рамках слоя. Следует отметить, что в дополнение к разбиению программного обеспечения на слои с указанием отношений между ними многослойное представление должно также конкретизировать возможные варианты взаимоотношений между модулями одного слоя (рис. 6.10).

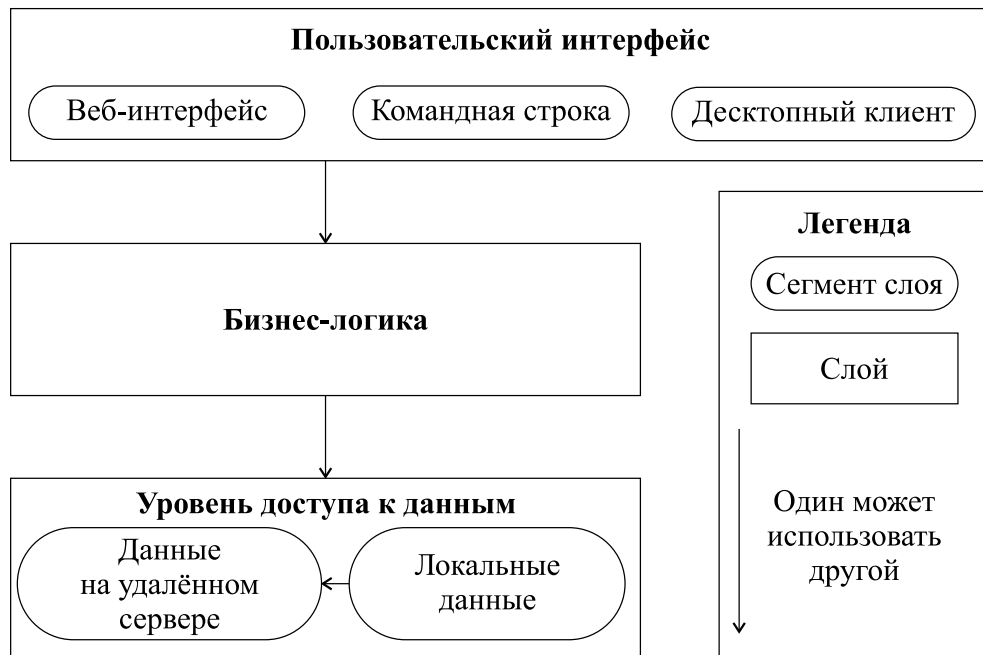


Рис. 6.10 – «Многослойный» архитектурный стиль с указанием возможных взаимодействий между модулями одного уровня

Рассмотрим более подробно понятие интерфейса. В рамках рассматриваемого архитектурного стиля под интерфейсом понимается не только программный интерфейс (API) в его классическом представлении, но и все допущения или предположения, которые имеются в более высокоуровневом слое по отношению к более низкоуровневому. Так, например, изменения в низкоуровневом слое, влияющие на производительность системы в целом, могут вызвать изменения в высокоуровневом слое, связанном с данным слоем посредством некоторого интерфейса.

Стиль «модель данных»

Моделирование структур данных широко используется при разработке как программного обеспечения в целом, так и, в особенности, автоматизированных информационных систем. Результатом моделирования является модель данных, описывающая основные сущности и связи между ними. В качестве нотации для построения модели данных обычно используются модель «сущность-связь» или диаграммы классов UML.

Моделирование структур данных выполняется в три последовательных этапа.

1. Концептуальная модель данных. В рамках построения концептуальной модели данных, как правило, выделяются основные сущности и их взаимосвязи. Результатом также может быть модель предметной области в нотации UML.
2. Логическая модель данных. На данном этапе выделяются все основные атрибуты сущностей, и происходит выбор технологий управления данными (таких как реляционные СУБД), а также выполняется нормализация отношений.
3. Физическая модель данных. Физическая модель данных содержит конкретные детали реализации логической модели, такие как первичные ключи, индексы, разбиение сущностей на разделы, а также может отражать различные аспекты оптимизации базы данных.

В заключение, следует отметить, что документация по архитектуре любой программной системы содержит, как правило, хотя бы одно модульное представление [36]. В то же время, несмотря на безусловную важность модульного представления архитектуры, оно, характеризуя лишь статические аспекты программного продукта, не содержит данных о взаимодействии компонентов программы в процессе ее выполнения и, таким образом, непригодно для анализа надежности, производительности и многих других важных нефункциональных характеристик системы, для оценки которых используются представления компонентов и коннекторов или представления распределения.

Тип представления распределения

В типе представления распределения (*allocation viewtype*) документируются способ и место сопоставления архитектуры программного обеспечения с окружающей средой [45]. Программы, базы данных и интерфейсы типа представления компонентов и коннекторов сопоставляются с оборудованием в стиле развертывания (*deployment*). Модули сопоставляются с системами с помощью стиля реализации (*implementation*). Модули сопоставляются с людьми (ролями, группами людей, рабочими группами, организациями), уполномоченными выполнять разработку, техническое обслуживание и эксплуатацию модулей, с помощью стиля распределения работ (*work assignment*).

Стиль развертывания

Элементами архитектурного представления в стиле развертывания являются *элементы программного обеспечения* из представления компонентов и контроллеров и *элементы окружающей среды*, к которым относятся компоненты аппаратных или вычислительных платформ: процессор, память, сетевое оборудование или характеристики. К основным отношениям представления относятся отношения *распределения*, которое отражает использование физического окружения программными компонентами в процессе выполнения, а также отношения *миграции*, *миграции с копированием* и *миграции времени выполнения* в случае динамического распределения ресурсов.

Отношение *миграции* устанавливается между программным элементом на одном процессоре и таким же программным компонентом на другом процессоре и указывает на возможность перемещения программного элемента между процессорами, при этом в один момент времени программный элемент существует только на одном процессоре.

В случае отношения *миграции с копированием*, в отличие от операции миграции, создается копия программного элемента, которая затем передается на другой процессор при сохранении оригинального элемента на первом процессоре.

Отношение *миграции времени выполнения* не изменяет принадлежности кода выполняющему операцию процессору, однако исполнение программы может передаваться между процессорами. Копия процесса может существовать более чем на одном процессоре, однако активным в каждый момент времени является только один такой процесс. Таким образом, выполнение процесса может передаваться между процессорами.

Важными характеристиками элементов окружающей среды являются такие физические характеристики, как параметры вычислительного процессора (частота, количество процессоров, объем памяти, скорость шины, размер кэша и скорость выполнения инструкций), параметры памяти (объем и скорость чтения/записи), полоса пропускания, характеристики отказоустойчивости оборудования.

Элементы программного обеспечения включают такие параметры, как:

- потребность в ресурсах. Например, *вычисления требуют выполнения 32 579 инструкций* в среднем или при нормальных условиях и т. д.;
- требования к ресурсам и ограничения, которые должны выполняться при любых условиях. Например, *определенная операция должна выполняться не более 0,1 секунды*;
- критический приоритет выполнения, например для приложений, которые должны выполняться постоянно;
- триггеры миграции, которые определяют условия срабатывания установленных отношений миграции.

Представление развертывания находят свое применение при оценке производительности, доступности, надежности и безопасности. Данное представление может быть использовано в процессе тестирования для понимания зависимостей времени выполнения программного обеспечения, а также для проведения интеграционного тестирования и оценки стоимости необходимого аппаратного обеспечения.

В качестве графической нотации для данного архитектурного стиля, помимо языков ADL, можно использовать неформальные нотации, в рамках которых аппаратное обеспечение обозначается общепринятыми иконками, или нотации диаграммы развертывания языка UML.

N-уровневая/3-уровневая архитектура

Традиционная 2-уровневая архитектура клиент-сервер имеет множество недостатков [37], включая тенденцию тесного связывания данных и бизнес-логики приложения на сервере, что может иметь негативное влияние на расширяемость и масштабируемость системы, и зависимость от центрального сервера, что негативно

сказывается на надежности системы. Для решения этих проблем архитектурный стиль клиент-сервер может быть дополнен 3-уровневым (или N-уровневым) стилем развертывания, описываемым ниже, в котором устранены некоторые недостатки, свойственные 2-уровневой архитектуре клиент-сервер, и обеспечиваются некоторые дополнительные преимущества.

N-уровневая и 3-уровневая архитектуры описывают разделение функциональности на сегменты, во многом аналогично многослойной архитектуре, но в данном случае эти сегменты могут физически размещаться на разных компьютерах, называемых уровнями [37].

Характеристиками N-уровневой архитектуры приложения являются функциональная декомпозиция приложения и распределенное развертывание сервисных компонент, что обеспечивает повышенную масштабируемость, доступность, управляемость и эффективность использования ресурсов. Каждый уровень абсолютно независим от всех остальных, кроме тех, с которыми он непосредственно соседствует. Так, компонентам уровня N требуется лишь знать, как обрабатывать запрос от компонентов уровня $(n + 1)$ и как передавать этот запрос на уровень $(n - 1)$ (если таковой имеется), а также как обрабатывать результаты запроса. Для обеспечения лучшей масштабируемости связь между уровнями обычно асинхронная. N-уровневая архитектура, как правило, имеет не менее трех отдельных логических частей, каждая из которых физически размещается на разных серверах. Каждая часть отвечает за определенную функциональность. При использовании многослойного подхода слой развертывается на уровне, если предоставляемая этим слоем функциональность используется более чем одним сервисом или приложением уровня.

Примерами архитектурного стиля могут служить типовое финансовое Веб-приложение с высокими требованиями к безопасности или типовой насыщенный клиент, в котором слой представления развернут на клиентских компьютерах, а бизнес-слой и слой доступа к данным развернуты на одном или более серверных уровнях.

Основными преимуществами N-уровневого архитектурного стиля являются:

- удобство поддержки. Уровни не зависят друг от друга, что позволяет выполнять обновления или изменения, не оказывая влияния на приложение в целом;
- масштабируемость. Уровни организовываются на основании развертывания слоев, поэтому масштабировать приложение довольно просто;
- гибкость. Управление и масштабирование каждого уровня может выполняться независимо, что обеспечивает повышение гибкости;
- доступность. Приложения могут использовать модульную архитектуру, которая позволяет использовать в системе легко масштабируемые компоненты, что повышает доступность системы.

Стиль распределения работ

В рамках данного стиля назначаются ответственные за реализацию системы люди или группы людей. Стиль распределения работ обычно используется для назначения исполнителей и позволяет удостовериться в том, что реализация и внед-

рение каждого модуля системы возложены на какую-либо команду или отдельного разработчика. Даже в случае, когда отдельный модуль приобретается по коммерческой лицензии, обычно необходимо тестирование, изучение и интеграция такого модуля в целостную программную систему.

Элементами данного стиля являются *модули* программного обеспечения и *организационные единицы* (команды разработчиков или отдельные разработчики). В качестве характеристик организационных единиц возможно указание их компетенций и свободного времени. В качестве основного отношения стиля используется отношение распределения — назначения исполнителей отдельным модулям.

Простой и удобной формой графической нотации распределения работ является табличная форма, в которой построчно модулям системы поставлены в соответствие исполнители.

Тип представления компонентов и коннекторов

В типе представления компонентов и коннекторов (*C&C — component-and-connector*) документируются процесс исполнения, хранилища данных, клиенты, серверы и взаимодействия, рассматриваются управляющая логика и потоки данных компонентов исполняемого приложения. Схемы компонентов и коннекторов представляют собой, как правило, отображения приложений в виде «рамки и линий» [37].

Составляющие элементы представления компонентов и коннекторов — это отображения клиентов, серверов, фильтров/преобразований, объектов и баз данных. Интерфейсы между компонентами называются портами — тем самым подчеркивается, что они относятся ко времени исполнения программы. Все порты должны быть явным образом документированы, как и коннекторы, определяющие вызовы процедур, вызовы базы данных, обмен сообщениями и асинхронные потоки данных. Значительный недостаток во многих диаграммах архитектуры компонентов и коннекторов — это отсутствие наименований коннекторов. Все элементы архитектуры должны быть явным образом идентифицированы и определены.

Поскольку топологии потоков данных и управляющей логики не всегда синхронны, они должны быть представлены в диаграмме независимо друг от друга. В сложных системах следует использовать различные диаграммы для представления потоков данных и управляющей логики, а сопоставление элементов моделей компонентов и коннекторов и элементов типа модульного представления делает информацию о компонентах и коннекторах более понятной [37]. Основными стилями архитектуры в типе представления компонентов и коннекторов являются следующие [45]:

- каналы и фильтры (*Pipe-and-filter*);
- совместно используемые данные (*Shared-data*);
- публикация-подписка (*Publish-subscribe* или *pub/sub*);
- клиент-сервер (*Client-server*);
- одноранговое взаимодействие (*Peer-to-peer*).

Архитектурная документация типа представления компонентов и коннекторов, как и остальная архитектурная документация, состоит из двух основных составляющих [36]: первичного представления, которое обычно представляет собой диа-

грамму в неформальной или полужформальной нотации, и сопровождающей документации, которая раскрывает содержимое первичного представления. Однако в случае представления компонентов и коннекторов сопровождающая документация выполняет особенно важную роль. Коннекторы, как правило, имеют очень сложную структуру, нетривиальную в реализации. В качестве примеров таких коннекторов можно привести коннекторы клиент-сервер, доступа к данным или подписки на события. Так, коннектор клиент-сервер обычно представляет протокол взаимодействия между клиентом и сервером и описывает механизмы установления соединения и управления клиентскими сессиями.

Следует отметить, что коннекторы могут соединять как два, так и более двух компонент одновременно, и в этом плане нотации архитектурного представления не накладывают каких-либо искусственных ограничений на архитектуру.

Отметим несколько ключевых особенностей графического представления архитектурного стиля компонентов и коннекторов.

- Графическое изображение выступает ключом для дополнительной документации, которая раскрывает все архитектурные аспекты отдельных элементов, их характеристик и отношений.
- Первичное представление должно содержать только данные, которые легко представить и прочесть в графической форме на одной диаграмме.
- Лексическое содержание первичного представления полностью раскрывается в соответствующем глоссарии дополнительной документации.
- Первичное представление отражает количество и типы интерфейсов как компонентов, так и коннекторов.
- Семантика и реализация как компонентов, так и коннекторов может носить сложный нетривиальный характер.

Ввиду перечисленных особенностей представления данного типа могут использоваться для раскрытия механизмов достижения нефункциональных требований: доступности, надежности, производительности и безопасности данных. При этом каждый нефункциональный аспект работы приложения подробно раскрывается в сопроводительной документации.

Таким образом, первичное представление архитектурной документации типа компонентов и коннекторов в совокупности с соответствующей сопроводительной документацией образуют базис, необходимый для обоснования всех основных решений в рамках архитектуры ПО и достижения заявленных нефункциональных требований. На основании такой документации возможно объяснение поведения системы во время выполнения и влияния отдельных архитектурных аспектов на показатели качества программного продукта.

Элементами представления компонентов и коннекторов являются компоненты и коннекторы.

Компонент — это блок обработки или хранения данных. Каждый компонент характеризуется определенным набором портов, через которые он взаимодействует с другими компонентами посредством коннекторов.

Коннектор определяет путь взаимодействия между компонентами. Каждый коннектор имеет определенную роль, которая определяет способ использования коннектора для организации взаимодействия компонентов.

Компоненты и коннекторы могут иметь отношения двух видов.

1. Вложение — отношение, при котором порты компонентов связываются с ролями коннекторов, образуя граф компонентов и коннекторов.
2. Делегирование интерфейса — отношение, при котором порты коннекторов связываются с одним или несколькими портами «субархитектуры» (аналогично для ролей коннекторов).

При построении диаграммы должны соблюдаться следующие ограничения:

- компоненты могут быть связаны только с коннекторами;
- коннекторы могут быть связаны только с компонентами;
- вложения возможны только между совместимыми портами и ролями;
- делегирование интерфейса возможно только между двумя совместимыми ролями (портами);
- коннектор не может быть изолирован. Любой коннектор должен быть связан с компонентом.

Компоненты представляют основные вычислительные элементы и хранилища данных, присутствующие во время выполнения программы. Каждый элемент диаграммы характеризуется наименованием, которое должно отражать предназначение данного элемента. Кроме того, наименования элементов позволяют соотносить графическую нотацию с соответствующими разделами сопроводительной документации. Каждый порт компонента определяет точку взаимодействия этого компонента с окружающей средой. Обычно порт имеет строго определенный тип, который описывает поведение и характер взаимодействия с коннектором. Один компонент может иметь несколько портов одного типа, что отличает их от интерфейсов в модульном представлении, которые никогда не повторяются в рамках одного модуля. Так, например, сервер может иметь несколько однотипных портов для обработки параллельных клиентских запросов. Нотациями представления компонентов и коннекторов допускается указание количества однотипных портов в фигурных скобках. Например, указание «[3]» рядом с портом свидетельствует о наличии трех однотипных портов у данного компонента, а «[0..10]» говорит о том, что количество портов может изменяться в пределах от 0 до 10 (рис. 6.11).

Следует отметить, что представление компонентов и коннекторов имеет очень важное отличие от модульного представления [36, 45]: в представлении компонентов и коннекторов отражаются сущности времени выполнения программы, в то время как элементами модульного представления являются единицы реализации программного кода. Так, например, система, состоящая из 10 идентичных клиентских приложений и одного сервера на диаграмме компонентов и коннекторов, будет представлена 11 компонентами и 10 коннекторами, но только двумя модулями в модульном представлении.

Для оформления диаграмм представления компонентов и коннекторов можно использовать как формальные нотации (ADL), так и полуформальную нотацию диаграммы компонентов языка UML. Пример оформления взаимодействия компонентов в нотации UML приведен на рисунке 6.12.

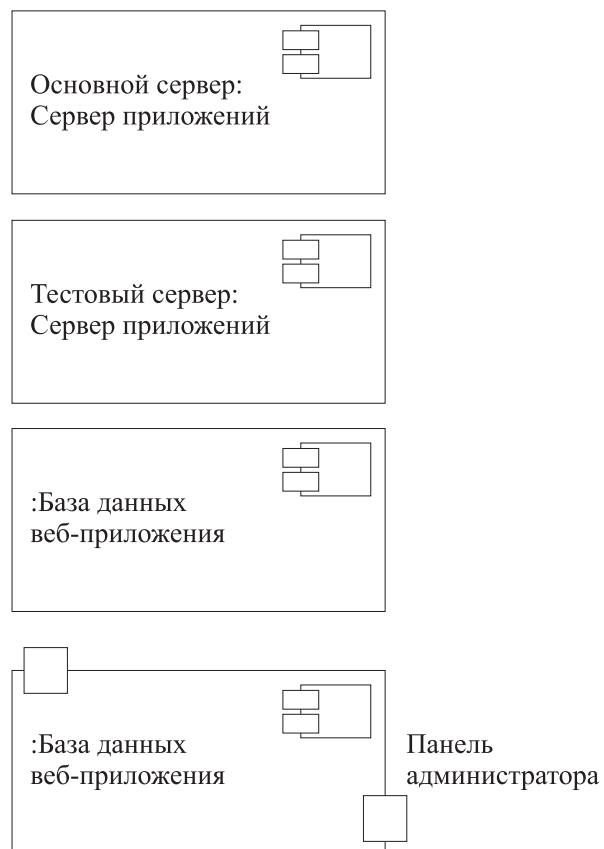


Рис. 6.11 – Примеры компонентов архитектурного стиля «компонентов и коннекторов»

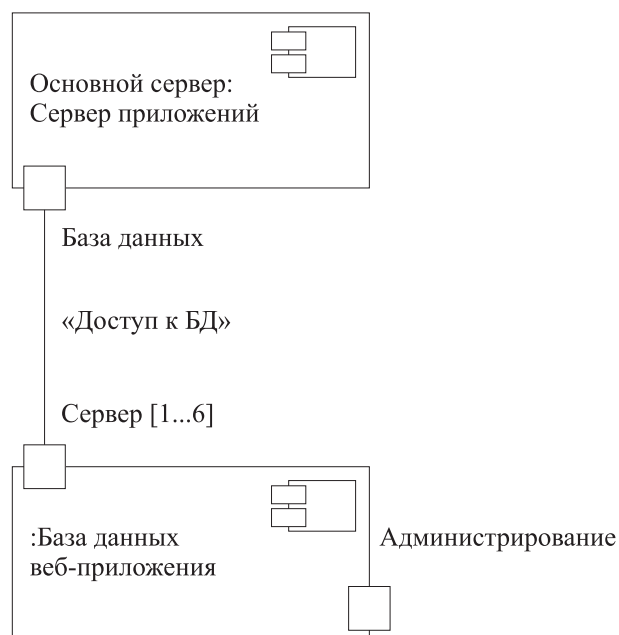


Рис. 6.12 – Пример диаграммы архитектурного стиля «компонентов и коннекторов»

Отметим также, что компоненты в представлении компонентов и коннекторов могут представлять сложные подсистемы, которые, в свою очередь, сами могут быть описаны с помощью представления компонентов и коннекторов. Архитектура такой подсистемы может быть раскрыта как внутри основной диаграммы, так и в отдельном документе. Отдельное внимание при документировании подсистем следует уделять соотношению внешних и внутренних портов компонента.

Стиль «поток данных»

В данной модели компоненты выступают в качестве преобразователей данных, а коннекторы выполняют роль передатчиков данных с выхода одного коннектора на вход другого [36, 46]. Каждый тип компонентов характеризуется определенным числом входящих и исходящих портов. Работа компонента сводится к получению данных на входе и подаче модифицированных данных на выход. Данный стиль используется в системах, выполняющих поточную обработку данных, которая может быть представлена в виде определенной последовательности шагов. Наиболее известным шаблоном архитектурного стиля «поток данных» является стиль «каналов и фильтров».

Данные поступают на входящий порт фильтра, преобразуются некоторым образом и подаются через канал на вход следующего фильтра. Один фильтр может получать данные с разного количества портов и передавать данные на множество портов. Современными примерами использования такой архитектуры являются системы обработки сигналов, веб-сервер Apache и системы, использующие конвейерную обработку данных в терминологии Unix [47].

Преобразование данных выполняется фильтрами, работающими одновременно и последовательно. Характеристики фильтра могут определять форматы входных/выходных данных, скорость обработки данных и тип выполняемых преобразований. Каналы, являющиеся коннекторами, выполняют функцию передачи данных с выхода одного фильтра на вход другого. Канал имеет один вход и один выход, не изменяет проходящие по нему данные и сохраняет их последовательность. Отношения на графе каналов и фильтров ставят в соответствие исходящий порт одного фильтра входу канала и входящий порт другого фильтра выходу канала.

Ограничения, накладываемые шаблоном каналов и фильтров:

- канал соединяет исходящий порт одного фильтра с входящим портом другого фильтра;
- соединенные каналом фильтры должны поддерживать один и тот же формат данных, передающихся по каналу;
- стиль шаблона может накладывать дополнительные ограничения на образование циклических связей (такой граф без циклических связей иногда называют конвейером).

Шаблон каналов и фильтров обычно используется для повышения независимости отдельных компонентов ПО и возможности их повторного использования, повышения производительности за счет параллельной обработки данных и упрощения поведения и алгоритмов обработки данных системы в целом.

Стиль «вызов-возврат» описывает модель программной системы, в которой отдельные компоненты предоставляют множество сервисов, которые могут быть ис-

пользованы другими компонентами [36]. Компонент, предоставляющий какой-либо сервис, блокируется до завершения его выполнения. Таким образом, стиль «вызов-возврат» является архитектурным аналогом вызова процедуры в программировании. Коннекторы отвечают за доставку запроса на вызов сервиса и возврат результата его выполнения. К стилю «вызов-возврат» относятся такие распространенные архитектурные шаблоны, как *Representational State Transfer* (REST), клиент-сервер и шаблон однорангового взаимодействия.

Архитектурный стиль «клиент-сервер»

Архитектурный стиль «клиент-сервер» описывает распределенные системы, состоящие из отдельных клиента и сервера и соединяющей их сети. Простейшая форма системы «клиент-сервер», называемая 2-уровневой архитектурой, — это серверное приложение, к которому напрямую обращается множество клиентов. Данный архитектурный стиль описывает ассиметричные отношения между клиентом и одним или более серверами, где клиент инициирует один или более запросов (возможно, с использованием графического пользовательского интерфейса), ожидает ответы и обрабатывает их при получении.

Элементами архитектурного стиля являются *клиенты*, инициирующие взаимодействие, *серверы*, предоставляющие набор сервисов для клиентов, и *коннекторы запрос/ответ*, используемые клиентами для организации запросов к серверу. Параметры коннектора определяют формат запросов и наличие или отсутствие шифрования данных. Основным отношением стиля «клиент-сервер» является отношение вложения, ассоциирующее порты запросов клиентов с ролями запросов сервера и роли ответов сервера с портами ответов клиента.

Ограничения, накладываемые архитектурным стилем:

- клиенты соединяются с серверами через коннекторы запрос/ответ;
- компоненты сервера могут являться клиентами для других серверов;
- дополнительно могут быть ограничены количество отношений для данного порта и возможность установления соединений между серверами;
- компоненты могут быть объединены в уровни (слои).

На сегодняшний день примерами архитектурного стиля «клиент-сервер» могут служить веб-приложения, выполняющиеся в Интернете или внутренних сетях организаций; настольные приложения, выполняющие доступ к сетевым сервисам обработки и хранилищам данных, например программы чтения электронной почты, FTP-клиенты и средства доступа к базам данных.

Выполнение запросов производится в синхронном режиме. Клиент после отправки запроса переходит в заблокированное состояние (состояние ожидания) до тех пор, пока сервер не передаст ответ на запрос. Клиент-серверный стиль может быть расширен за счет разрешения серверу инициировать отдельные взаимодействия или введения механизма *сессий* для управления соединениями.

К преимуществам архитектурного стиля «клиент-сервер» можно отнести следующие [37].

- Высокий уровень безопасности, который достигается за счет того, что хранение и обработка данных выполняются на стороне сервера, который обыч-

но обеспечивает больший контроль безопасности, чем клиентские компьютеры.

- Централизованный доступ к данным. Поскольку данные хранятся только на сервере, администрирование доступа к данным и поддержание их актуального состояния проще, чем в других архитектурных стилях.
- Простота обслуживания. Роли и ответственность вычислительной системы, как правило, распределены между несколькими серверами, общающимися друг с другом по сети. Благодаря этому клиент гарантированно остается неосведомленным и не подверженным влиянию событий, происходящих с сервером (ремонт, обновление либо перемещение).

Архитектурный стиль однорангового взаимодействия

В рамках данного стиля компоненты архитектуры могут взаимодействовать друг с другом напрямую, обмениваясь предоставляемыми сервисами. Основным отличием от стиля «клиент-сервер» является отсутствие асимметричности во взаимодействии: любой компонент может инициировать запрос сервиса другого компонента [36, 37]. Коннекторы в стиле однорангового взаимодействия могут представлять комплексные протоколы двунаправленного взаимодействия. Примерами реализации данного архитектурного стиля являются файлообменные сети, такие как BitTorrent, системы IP-телефонии, а также системы распределенных вычислений.

Конкретные реализации однорангового взаимодействия могут накладывать ограничения на количество одновременных соединений для отдельных портов и видимость участников сети друг другу, а также определять компоненты, ответственные за маршрутизацию, индексирование и поиск участников сети.

К преимуществам данного стиля можно отнести повышенную доступность и масштабируемость программных средств [36].

Сервис-ориентированная архитектура

В сервис-ориентированной архитектуре (СОА) программное средство состоит из множества распределенных компонентов, предоставляющих или потребляющих различные сервисы [36, 45]. Компоненты, предоставляющие и использующие определенный сервис, могут быть реализованы на разных программно-аппаратных платформах и являются самостоятельными приложениями, зачастую входящими в состав разных программных продуктов или даже принадлежащими различным организациям.

Основными компонентами стиля являются поставщики и потребители сервисов. Кроме того, в СОА в качестве посредников для организации и управления инфраструктурой могут применяться следующие дополнительные специализированные компоненты.

- Шина сервисов предприятия, отвечающая за доставку сообщений между потребителями и поставщиками сервисов.
- Регистр сервисов, который может быть использован для улучшения поиска поставщиков сервисов.

- Сервер оркестровки (оркестровка — это автоматическое размещение, координация и управление сложными компьютерными системами и службами), — специальный компонент, который отвечает за выполнение определенных скриптов в случае наступления заранее определенных событий и служит для организации взаимодействия между компонентами СОА.

К основным коннекторам стиля СОА относятся коннекторы «вызов-возврат», использующиеся, как правило, в рамках протокола HTTP, и коннекторы асинхронного обмена сообщениями, использующими различные протоколы «точка-точка» или «публикация-подписка».

К преимуществам архитектуры СОА можно отнести возможности включения в состав программной системы компонентов для разных платформ, реализованных на разных языках программирования, интеграции с устаревшими и неподдерживаемыми программными системами, а также динамической конфигурации такой системы.

Стили, основанные на событиях

Стили, основанные на событиях, позволяют компонентам обмениваться информацией посредством асинхронных сообщений. Такие системы часто проектируются как набор слабо связанных компонентов, которые управляют поведением друг друга с помощью триггеров. Сегодня существует и применяется на практике множество событийных стилей. Некоторые из них предполагают наличие соединений «точка-точка», благодаря чему система функционирует схожим образом с системами «вызов-возврат», но допуская при этом большую производительность и параллельную обработку запросов благодаря неблокирующей природе асинхронного взаимодействия. Одним из наиболее распространенных стилей в рамках данной парадигмы является стиль «публикация-подписка» [36].

Для стиля «публикация-подписка» характерно взаимодействие между компонентами за счет объявления событий. Отдельные компоненты могут быть подписаны на множество событий других компонентов. Инфраструктура такой системы обеспечивает доставку сообщений всем подписанным на событие компонентам. Таким образом, основной формой коннекторов является шина событий. Компоненты, объявляя события, помещают их в шину, которая затем доставляет сведения о событии всем заинтересованным адресатам. Модель такой архитектуры можно представить в виде набора независимых процессов, которые реагируют на события, генерируемые их окружением, что, в свою очередь, посредством новых объявленных событий может вызвать ответную реакцию в других компонентах.

В качестве примеров архитектуры «публикация-подписка» можно привести следующие:

- графический пользовательский интерфейс, в котором действия пользователя обрабатываются как события соответствующими обработчиками;
- приложения Модель-Представление-Контроллер (MVC), в которых представление получает уведомления об изменениях в соответствующей модели;
- социальные сети, в которых предусмотрено получение уведомлений об активности «друзей».

Стили, основанные на хранилище данных

Основные функции системы, в которых применяются архитектурные стили данной группы, связаны с хранением, обработкой и представлением больших количеств данных [36]. Такие системы включают в себя один или несколько компонентов, называемых хранилищами данных, которые выполняют функции постоянного хранения данных, и другие компоненты, которые могут выполнять операции чтения и записи данных в хранилище. Зачастую, доступ к хранилищу обеспечивается специальным программным обеспечением — системами управления базами данных (СУБД), которые предоставляют интерфейс «вызов-возврат» для операций чтения и управления данными. Кроме того, СУБД обычно предоставляют сервисы обработки транзакций и поддержания целостности данных, функции обеспечения безопасности и контроля доступа. В рамках архитектурного стиля компонентов и коннекторов хранилище обычно представляет собой совокупность СУБД и репозитория данных.

Выделяют две основные группы архитектурных стилей, основанных на использовании хранилища данных: стиль «классной доски» и стиль совместного доступа к данным [47].

В стиле совместного доступа к данным инициаторами действий с данными выступают компоненты, использующие эти данные. Порядок работы определяется только потоком внешних событий. Выделяется общее хранилище данных — репозиторий. Каждый обработчик запускается в ответ на соответствующее ему событие и преобразует часть данных в репозитории. В системах, построенных по принципу «классной доски», ответственность за уведомление об изменениях в хранилище данных других заинтересованных компонентов системы возлагается на само хранилище данных. Реализация архитектурного стиля «классной доски» возможна благодаря механизму триггеров, активируемых при добавлении, изменении или удалении данных. Сегодня встроенная поддержка триггеров присутствует в большинстве современных реляционных СУБД.

Стиль совместного доступа к данным предусматривает два вида компонентов: хранилища данных и компоненты доступа к данным, а также два вида коннекторов: коннекторы чтения и записи данных. Компоненты доступа к данным выполняют операции, требующие использования данных одного или нескольких хранилищ. Результат таких операций также заносится в одно или несколько хранилищ данных. Вся обработка данных выполняется только компонентами доступа к данным. В некоторых случаях допускается взаимодействие компонентов доступа к данным друг с другом. Хранилища данных при такой архитектуре отвечают за обработку конкурирующих запросов, сохранение целостности данных, управление кэшированием и отказоустойчивость системы.

Стиль совместного доступа к данным применяется в случае наличия различных поставщиков и потребителей данных и обеспечивает хорошую модифицируемость системы [36]. Такой архитектурный шаблон находит широкое распространение в клиент-серверных и многоуровневых системах и хорошо сочетается со стилем «модель данных», отражающим внутреннюю структуру данных. В качестве хранилища, как правило, используется реляционная СУБД, драйвера которой включаются в состав клиентского приложения.

6.5 Комплексный подход к построению архитектуры

В заключение, следует отметить, что архитектура программной системы практически никогда не ограничена лишь одним архитектурным стилем, зачастую она является сочетанием архитектурных стилей, образующих полную систему [37]. Например, может существовать SOA-дизайн, состоящий из сервисов, при разработке которых использовались многослойная архитектура и объектно-ориентированный архитектурный стиль. Сочетание архитектурных стилей также полезно при построении Интернет Веб-приложений, где можно достичь эффективного разделения функциональности за счет применения многослойного архитектурного стиля.

Таким образом, можно отделить логику представления от бизнес-логики и логики доступа к данным. Требования безопасности организации могут обуславливать либо 3-уровневое развертывание приложения, либо развертывание с более чем тремя уровнями. Уровень представления может развертываться в пограничной сети, располагающейся между внутренней сетью организации и внешней сетью. В качестве модели взаимодействия на уровне представления может применяться шаблон представления с отделением (разновидность многослойного стиля), такой как Model-View-Controller (MVC). Также можно выбрать архитектурный стиль SOA и реализовать связь между Веб-сервером и сервером приложений посредством обмена сообщениями. Создавая настольное приложение, можно реализовать клиент, который будет отправлять запросы к программе на сервере. В этом случае развертывание клиента и сервера можно выполнить с помощью архитектурного стиля «клиент-сервер» и использовать компонентную архитектуру для дальнейшего разложения дизайна на независимые компоненты, предоставляющие соответствующие интерфейсы. Применение объектно-ориентированного подхода к этим компонентам повысит возможности повторного использования, тестирования и гибкость [37].



Контрольные вопросы по главе 6

1. Что такое архитектура программного обеспечения? Чем архитектура отличается от структуры?
2. Зачем нужно разрабатывать архитектуру программной системы? Зачем документировать результаты разработки?
3. Какие нотации документации по архитектуре ПО Вам известны? Можно ли использовать диаграммы UML для документирования архитектуры ПО?
4. Какие принципы проектирования архитектуры Вам известны?
5. Что такое архитектурный стиль? Какие типы архитектурных стилей Вам известны?
6. Перечислите основные стили в рамках типа модульного представления архитектуры?

7. Перечислите основные архитектурные стили в рамках типа представления компонентов и коннекторов? Что такое клиент-серверная архитектура?
8. Перечислите основные архитектурные стили в рамках типа представления распределения? Зачем используется стиль распределения работ?
9. Что такое многозвенная (многоуровневая) и многослойная архитектура? Чем отличаются эти понятия?

ЗАКЛЮЧЕНИЕ

Представленный в пособии обзор всех основных аспектов проектирования программных средств и их архитектуры позволяет разрабатывать современные программные системы, используя актуальные методики управления жизненным циклом и требованиями ПП. Такой подход позволит Вам достичь необходимых функциональных и нефункциональных показателей качества системы.

Рассмотренные архитектурные стили позволяют создать хорошую основу современной информационной системы, а методы объектно-ориентированного проектирования помогут выбрать и реализовать наиболее эффективную структуру программного комплекса.

В качестве источника дополнительных сведений по вопросам проектирования программных средств авторы рекомендуют использовать публикации и издания, указанные в списке литературы настоящего пособия.

Отметим в заключение, что наиболее подверженными изменениям среди рассмотренных в настоящем пособии являются вопросы стандартов менеджмента качества программного обеспечения. Актуальную информацию о последних стандартах можно получить из ГОСТов или на сайте международной организации по стандартизации ISO.

ЛИТЕРАТУРА

- [1] Орлик С. Основы программной инженерии (по SWEBOK) [Электронный ресурс] / С. Орлик, Д. Буруй // Сайт «Основы программной инженерии». — URL : http://lukovsky.dynalias.net/study/x513/8%20семестр/PIS/lec/swebok/7-software_engineering_management.pdf (дата обращения: 09.07.2015).
- [2] Brown S. Software Architecture for Developers [Электронный ресурс] // Leanpub. — URL : <https://leanpub.com/software-architecture-for-developers> (дата обращения: 14.01.2015).
- [3] ГОСТ 28806—90. Качество программных средств. Термины и определения. — М., 1990. — 8 с.
- [4] ГОСТ 19781—90. Обеспечение систем обработки информации программное. Термины и определения. — (взамен ГОСТ 19781—83, ГОСТ 19.004—80 ; введ. 27.08.90). — М. : Стандартинформ, 2010. — 16 с.
- [5] Гагарина Л. Г. Технология разработки программного обеспечения : учеб. пособие / Л. Г. Гагарина, Е. В. Кокорева, Б. Д. Виснадул // М. : ИД «Форум», 2008. — 400 с.
- [6] Гриценко Ю. Б. Операционные среды, системы и оболочки / Ю. Б. Гриценко // Томск : Томский межвузовский центр дистанционного образования, 2005. — 281 с.
- [7] ГОСТ ISO 9000—2011. Системы менеджмента качества. Основные положения и словарь. — М. : Стандартинформ, 2012. — 32 с.
- [8] ГОСТ Р ИСО/МЭК 9126—93. Информационная технология. Оценка программной продукции. Характеристики качества и руководства по их применению. — Введ. 28.12.93. — М. : ИПК Издательство стандартов, 2004. — 12 с.
- [9] Кросби Ф. Б. Качество бесплатно / Ф. Б. Кросби. — Нью-Йорк : МакГро-Хилл, 1979.
- [10] Жарко Е. Ф. Сравнение моделей качества программного обеспечения: аналитический подход / Е. Ф. Жарко // XII Всероссийское совещание по проблемам управления (ВСПУ—2014). — 2014. — С. 4585–4594.

- [11] ISO 9000. Менеджмент качества [Электронный ресурс] // ИСО — Международная организация по стандартизации. — URL: http://www.iso.org/iso/ru/iso_9000 (дата обращения: 14.01.2015).
- [12] ISO — Стандарты ИСО — ISO/IEC JTC 1/SC 7 — Разработка программного обеспечения и системотехника [Электронный ресурс] // ИСО — Международная организация по стандартизации. — URL : http://www.iso.org/iso/ru/home/store/catalogue_tc/catalogue_tc_browse.htm?commid=45086&published=on&inclusesc=true (дата обращения: 14.01.2015).
- [13] McCall J. A. Factors in Software Quality: Concept and Definitions of Software Quality. Final Technical Report / J. A. McCall, P. K. Richards, G. F. Walters. — National Technical Information Service, Springfield, 1977. — Vol. 1.
- [14] McCall J. A. Factors in Software Quality: Metric Data Collection and Validation. Final Technical Report / J. A. McCall, P. K. Richards, G. F. Walters. — National Technical Information Service, Springfield, 1977. — Vol. 2.
- [15] McCall J. A. Factors in Software Quality: Metric Data Collection and Validation. Final Technical Report / J. A. McCall, P. K. Richards, G. F. Walters. — National Technical Information Service, Springfield, 1977. — Vol. 3.
- [16] Characteristics of Software Quality, TRW Series of Software Technology / B. W. Boehm [and others]. — Amsterdam, North Holland, 1978. — 166 с.
- [17] Grady R. B. Software Metrics: Establishing a Company-Wide Program / R. B. Grady, D. L. Caswell. — Prentice-Hall, 1987. — 275 с.
- [18] IEEE SA 610.12—1990—IEEE Standard Glossary of Software Engineering Terminology [Электронный ресурс] // IEEE Standards association— URL : <http://standards.ieee.org/findstds/standard/610.12-1990.html> (дата обращения: 14.01.2015).
- [19] ГОСТ Р ИСО/МЭК 12207—2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств. — Введ. 01.03.12. — М. : Стандартинформ, 2011. — 105 с.
- [20] ISTQB Exam Certification—Study material for Foundation level, through ISTQB and ASTQB Exam, Certification questions, answers, tutorials and more. — URL : <http://istqbexamcertification.com> (дата обращения: 25.12.2014).
- [21] Royce W. Management the development of large software systems [Электронный ресурс] // UMD Department of computer science. — URL : <http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf> (дата обращения: 14.01.2015).
- [22] Sommerville I. Requirements Engineering: A Good Practice Guide / Ian Sommerville, Pete Sawyer // Wiley. — 404 с.

- [23] Вигерс Карл. Разработка требований к программному обеспечению. / Карл Вигерс, Джой Битти. — 3-изд., доп. — М. : Русская редакция ; СПб. : БХВ-Петербург, 2014. — 736 с.
- [24] Business Rules Group, Defining Business Rules ~ What Are They Really? (formerly known as the «GUIDE Business Rules Project Report» 1995). — 4th ed. — США, 2000.
- [25] Systems and software engineering — Life cycle processes — Requirements engineering [Электронный ресурс] // Webstore. International Electrotechnical Commission — URL : http://webstore.iec.ch/preview/info_isoiecieee29148%7Bed1.0%7Den.pdf (дата обращения: 18.01.2015).
- [26] Robertson S. Mastering the Requirements Process: Getting Requirements Right / S. Robertson, J. Robertson // Addison-Wesley, 2012. — 600 с.
- [27] Davis A. Just Enough Requirements Management: Where Software Development Meets Marketing // Dorset House Publishing Co., Inc. New York, NY, USA, 2005.
- [28] Орлов С. А. Технологии разработки программного обеспечения / С. А. Орлов, Б. Я. Цилькер. — 4-е изд. — СПб. : Питер, 2012. — 608 с.
- [29] Леоненков А. В. Самоучитель UML / А. В. Леоненков. — СПб. : БХВ-Петербург, 2001. — 304 с. : ил.
- [30] Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений / Буч Г. — 3-е изд. — М. : Вильямс. — 721 с.
- [31] Гома Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений / Х. Гома. — М. : ДМК, 2011. — 699 с.
- [32] Рамбо Дж. UML 2.0 Объектно-ориентированное моделирование и разработка / Дж. Рамбо. — СПб. : Питер, 2007. — 545 с.
- [33] Обзор проектирования архитектуры программного обеспечения [Электронный ресурс] // MSDN. — URL : <http://msdn.microsoft.com/ru-ru/hh144976.aspx> (дата обращения: 19.01.2015).
- [34] Гриценко Ю. Б. Архитектура предприятия : учеб. пособие / Ю. Б. Гриценко. — Томск : Эль Контент, 2011. — 206 с.
- [35] Eeles P. What is a software architecture [Электронный ресурс] // IBM developer networks. — URL: <http://www.ibm.com/developerworks/rational/library/feb06/eeles/> (дата обращения: 19.01.2015).
- [36] Documenting software architecture. Views and beyond. Second edition / Clements P. [and otherst]. — Addison Wesley, 2010. — 582 с.
- [37] Руководство Microsoft по разработке архитектуры программного обеспечения — 2-е изд. — Microsoft, 2014. — 529 с.

- [38] Community Software Architecture Definitions [Электронный ресурс] // Software ingeneering institute — URL : <http://www.sei.cmu.edu/architecture/start/glossary/community.cfm> (дата обращения: 19.01.2015).
- [39] Perry D. Foundations for the study of software architecture [Электронный ресурс] / D. Perry, A. Wolf. — URL : <http://users.ece.utexas.edu/~perry/work/papers/swa-sen.pdf> (дата обращения: 19.01.2015).
- [40] Garlan D. Introduction to the Special Issue on Software Architecture / D. Garlan, D. Perry // IEEE Transactions on Software Engineering. — 1995. — 21 (4). — P. 269–274.
- [41] Grady Booch. The Unified Modeling Language User Guide / Booch Grady, James Rumbaugh, Ivar Jacobson. — 2nd ed. — Addison-Wesley, 2005.
- [42] Bass L. Software Architecture in Practice / L Bass, P. Clements, R. Kazman. — 2nd ed. — Addison-Wesley, 2005.
- [43] Taylor R. Software Architecture: Foundations, Theory, and Practice / R. Taylor, N. Medvidovic, E. M. Dashofy. — Wiley, 2009.
- [44] Компьютерное общество IEEE: Рекомендуемые IEEE-методы описания архитектуры преимущественно-программных систем: стандарт IEEE 1472000, 2000.
- [45] Хилл Р. Сравнение многоуровневых и клиент-серверных систем [Электронный ресурс] // MSDN, 2007. — URL : <http://msdn.microsoft.com/ru-ru/library/ee895340.aspx> (дата обращения: 19.01.2015).
- [46] Pipes and Filters Pattern [Электронный ресурс] // MSDN. — URL : <http://msdn.microsoft.com/ru-ru/library/dn568100.aspx> (дата обращения: 19.01.2015).
- [47] Кулямин В. Компетентный подход в программировании [Электронный ресурс] // НОУ Интуит. — URL : <http://www.intuit.ru/studies/courses/64/64/lecture/1878?page=4><http://www.intuit.ru/studies/courses/64/64/lecture/1878?page=4> (дата обращения: 19.01.2015).

СПИСОК УСЛОВНЫХ ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ

БД — база данных

ЖЦ — жизненный цикл

ЖЦ ПО — жизненный цикл программного обеспечения

ООАП — объектно-ориентированный анализ и проектирование

ПО — программное обеспечение

ПП — программный продукт

ПС — программное средство

СУБД — система управления базами данных

ADL — язык описания архитектуры (*Architecture Description Language*)

CASE — *Computer-Aided System Engineering*

ISO — международная организация по стандартизации (*International Organization for Standardization*)

IEEE — институт инженеров электриков и электронщиков (*Institute of Electrical and Electronics Engineers*)

UML — унифицированный язык моделирования (*Unified Modeling Language*)

ГЛОССАРИЙ

Архитектура программного средства (системы) — это структура системы, включающая программные элементы, видимые извне свойства этих элементов и взаимоотношения между ними.

Архитектурный стиль — это набор принципов, высокоуровневая схема, обеспечивающая абстрактную инфраструктуру для семейства систем.

Жизненный цикл программного обеспечения — период времени, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации.

Качество — это весь объем признаков и характеристик продукции или услуги, который относится к их способности удовлетворять установленным или предполагаемым требованиям.

Нефункциональные требования — требования, которые определяют критерии работы системы в целом, а не отдельные сценарии поведения. Нefункциональные требования определяют системные свойства, такие как производительность, удобство сопровождения, расширяемость, надежность, средовые факторы эксплуатации.

Оркестровка — автоматическое размещение, координация и управление сложными компьютерными системами и службами.

Программное обеспечение — это совокупность программ системы обработки информации и программных документов, необходимых для эксплуатации этих программ.

Программное средство (англ. software) — это объект, состоящий из программ, процедур, правил, а также, если предусмотрено, сопутствующих им документации и данных, относящихся к функционированию системы обработки информации.

Система — это набор компонентов, объединенных для выполнения определенной функции или набора функций.

Системы управления базами данных — совокупность программных и лингвистических средств общего или специального назначения, которые обеспечивают организацию и хранение локальных баз данных (БД) на автономно работающих компьютерах либо централизованное хранение БД на файл-сервере и сетевой доступ к ним.

Требования — это спецификация того, что должно быть реализовано. В них описано поведение системы, свойства системы или ее атрибуты. Требования могут служить ограничениями в процессе разработки системы.

Функциональные требования — требования, которые определяют действия, которые система должна быть способной выполнить, связь входа/выхода в поведении системы.

Языки описания архитектуры — нотации, которые используются для формального описания характерно использование однозначной (как правило, математически описанной) семантики, благодаря чему становится возможным формальный анализ как синтаксиса описания, так и его семантики.

CASE-средства — программные комплексы, автоматизирующие весь технологический процесс анализа, проектирования, разработки и сопровождения сложных программных систем.

FURPS — классификация требований к программным системам. Название образовано от первых букв слов *functionality* (функциональные требования), *usability* (требования к удобству использования), *reliability* (требования к надежности), *performance* (требования к производительности), *supportability* (требования к поддержке).

Учебное издание

Милихин Михаил Михайлович

Рычагов Михаил Михайлович

ПРОЕКТИРОВАНИЕ И АРХИТЕКТУРА
ПРОГРАММНЫХ СРЕДСТВ

Учебное пособие

Корректор Осипова Е. А.

Компьютерная верстка Морозова Ю. В.

Издано в Томском государственном университете
систем управления и радиоэлектроники.
634050, г. Томск, пр. Ленина, 40
Тел. (3822) 533018.