

Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

А. А. Калентьев, Д. В. Гарайс, А. Е. Горяинов

---

---

# **НОВЫЕ ТЕХНОЛОГИИ В ПРОГРАММИРОВАНИИ**

---

---

Учебное пособие

Томск  
«Эль Контент»  
2014

УДК 004.413.002(075.8)

ББК 32.973.2-018я73

К 171

Рецензенты:

**Хабибулина Н. Ю.**, канд. техн. наук, доцент кафедры компьютерных систем  
в управлении и проектировании ТУСУРа;

**Самуилов А. А.**, генеральный директор компании по разработке мобильных  
приложений ООО «Арвью».

**Калентьев А. А.**

К 171 Новые технологии в программировании : учебное пособие / А. А. Калентьев, Д. В. Гарайс, А. Е. Горяинов — Томск : Эль Контент, 2014. — 176 с.

ISBN 978-5-4332-0185-9

В пособии рассматриваются современные технологии, используемые в процессе разработки программных систем. Определяется суть разработки ПО, выделяются основные этапы разработки и описываются необходимые результаты каждого этапа разработки ПО. Излагаются особенности командной разработки, описываются командные роли и методологии, используемые для облегчения работы в команде. В пособии также приводятся подходы и инструменты для улучшения качества, как самого программного кода, так и программной архитектуры в процессе реализации системы. В заключении приводятся инструменты, позволяющие эффективно организовать процесс разработки. Пособие будет полезно студентам специальностей и направлений подготовки, связанных с разработкой программных систем.

УДК 004.413.002(075.8)

ББК 32.973.2-018я73

ISBN 978-5-4332-0185-9

© Калентьев А. А.,  
Гарайс Д. В.,  
Горяинов А. Е., 2014

© Оформление.  
ООО «Эль Контент», 2014

# ОГЛАВЛЕНИЕ

<b>Введение</b>	<b>5</b>
<b>1 Процесс создания программного обеспечения</b>	<b>9</b>
1.1 Метафоры при создании ПО . . . . .	10
1.2 Этапы разработки ПО . . . . .	16
<b>2 Техническое задание</b>	<b>25</b>
2.1 Составление технического задания . . . . .	27
<b>3 Командные роли</b>	<b>34</b>
3.1 Командные роли по Белбину . . . . .	35
3.2 Функциональные роли . . . . .	37
<b>4 Методологии разработки ПО</b>	<b>40</b>
4.1 Что такое методология разработки ПО и зачем она нужна? . . . . .	40
4.2 Используемые методологии ПО . . . . .	42
4.2.1 Водопадная методология . . . . .	43
4.2.2 Гибкие методологии . . . . .	45
4.2.3 Другие методологии . . . . .	57
<b>5 Пользовательские интерфейсы</b>	<b>60</b>
5.1 Правила верстки пользовательского интерфейса . . . . .	61
5.2 Шаблоны пользовательского поведения . . . . .	66
5.3 Прототипирование . . . . .	70
<b>6 Документация</b>	<b>74</b>
6.1 Описание IDEF . . . . .	75
6.1.1 IDEF0 . . . . .	75
6.1.2 IDEF3 . . . . .	78
6.2 Unified Modeling Language . . . . .	80
6.2.1 Диаграмма деятельности . . . . .	82
6.2.2 Диаграмма вариантов использования . . . . .	84
6.2.3 Диаграмма классов . . . . .	87
6.2.4 Диаграмма пакетов . . . . .	92
6.3 Блок-схемы . . . . .	94
<b>7 Техники написания и поддержки кода</b>	<b>102</b>
7.1 Паттерны проектирования . . . . .	102
7.1.1 Порождающие паттерны . . . . .	104
7.1.2 Структурные паттерны . . . . .	117

---

7.2	Антипаттерны . . . . .	133
7.3	Оформление кода . . . . .	137
7.4	Рецензирование кода . . . . .	139
7.5	Рефакторинг . . . . .	141
7.6	Оптимизация . . . . .	143
<b>8</b>	<b>Тестирование</b>	<b>146</b>
8.1	Что такое тестирование? . . . . .	147
8.2	Тестовые случаи . . . . .	148
8.3	Классификация тестов . . . . .	150
8.4	Блочное тестирование . . . . .	154
<b>9</b>	<b>Информационное обеспечение процесса разработки</b>	<b>156</b>
9.1	Система управления проектами . . . . .	156
9.2	Системы контроля версий . . . . .	158
9.3	Непрерывная интеграция . . . . .	161
	<b>Заключение</b>	<b>165</b>
	<b>Литература</b>	<b>166</b>
	<b>Список условных обозначений и сокращений</b>	<b>170</b>
	<b>Глоссарий</b>	<b>171</b>

---

# ВВЕДЕНИЕ

---

Учебное пособие по курсу «Новые технологии в программировании» предназначено для студентов программистских специальностей и направлений подготовки. Следует сказать, что представленный в пособии материал будет полезен больше новичкам в программировании, чем специалистам с большим опытом, однако из-за большого охвата материала некоторые главы всё-таки смогут удивить и искушённого читателя.

Учебное пособие не является истиной в последней инстанции, т. к. нельзя в одну небольшую книгу уложить подробное изложение материала, необходимое для студента-программиста. Вполне возможно, что на момент прочтения пособия уже появятся новые методы, методики и технологии в программировании, по сравнению с которыми представленный материал покажется некорректным. Авторы вполне допускают подобное развитие событий, и их наступление является даже наиболее вероятным. Это обусловлено стремительным развитием IT-индустрии и выражено как в новых технологиях программирования (распределённые вычисления, кластерные вычисления), так и в новых платформах для разработки (появление мобильных устройств, таких как планшеты, смартфоны и пр.; переход инфраструктуры предприятий в облачные сервисы).

Для успешного освоения материала читатель должен обладать знаниями в области объектно-ориентированного программирования и проектирования систем, уметь пользоваться одним из объектно-ориентированных языков, оптимальным было бы знание C-подобных языков: C++, C#, Java и пр. В рамках курса НТВП для описания примеров будет использоваться язык C#.

Изучать пособие можно несколькими способами. Для новичков в программировании рекомендуется ознакомиться со всеми главами в представленном порядке, т. к. рассматриваются все стадии разработки ПО и инструментарий, для этого необходимый. В случае большего интереса к практическим аспектам написания кода, а не к организации инфраструктуры разработки, рекомендуется начинать с главы 7, а потом уже в интересующем порядке ознакомиться с другими главами.

Вкратце, о чём же это учебное пособие?

В главе 1 рассмотрены этапы создания программного обеспечения, приведены примеры, облегчающие восприятие разработки ПО. Обязательная глава для начинающих разработчиков, т. к. не все до конца понимают, что из себя представляет процесс создания ПО, иногда его даже романтизируют. Никакой романтики,

часто всё прагматично и жёстко, особенно при работе с заказчиком. Однако при жёсткости сроков и ресурсов, сам процесс разработки (кодирование, тестирование, отладка) являет очень творческим и интересным, что должно убедить читателя в правильности выбора профессии.

Глава 2 рассказывает о необходимости составления технического задания к проекту. В ней также приведены основные пункты, которые должны содержать техническое задание к программе. Данная глава будет полезна для понимания важности составления технического задания, потому что при его низком качестве или полном отсутствии ТЗ (вы будете думать, что это техническое задание) может легко превратиться в Точку Зрения заказчика.

В главе 3 описаны основные командные роли в проекте, т. е. кто и за что отвечает в процессе разработки. Так как даже средние проекты не делаются в одиночку, читателю будет полезно знать об основных командных ролях. В контексте тех или иных методологий разработки роли будут представлены в главе 4.

Методологии разработки, т. е. учения о методах, методиках, способах и средствах разработки ПО, приведены в главе 4. Изучение этой главы позволит читателю сложить мнение об особенностях используемых сегодня методологий разработки ПО. Помимо этого, в главе приводятся преимущества и недостатки тех или иных методологий, что позволит читателю при работе в реальном проекте предлагать их для повышения эффективности разработки или быстро встраиваться в процесс разработки, обладая знаниями о существующих методологиях.

Глава 5 описывает основные этапы разработки пользовательского интерфейса программы, что является одним из важнейших этапов, т. к. интерфейс, при его наличии, это то, что увидит пользователь, и на основе чего, скорее всего, он будет делать вывод о вашей программе. Эта глава поможет читателю формально подойти к разработке пользовательских интерфейсов, получив инструментарий и чёткие рекомендации, что нужно и что не нужно делать.

В главе 6 приводится описание нотаций для документации ПО: IDEF, UML, ЕСПД (ГОСТ 19.XXX-XX). Представлены основные аспекты разработки, которые могут быть задокументированы с помощью той или иной нотации. Для читателя данная глава может послужить хорошим стартом в изучении имеющихся способов документации разработки ПО, из-за сокращённого, по сравнению с первоисточниками, объёма излагаемого материала.

Глава 7 является наиболее объёмной во всём учебном пособии и имеет наибольшую практическую ценность для разработчика. В главе подробно с примерами разобраны основные паттерны проектирования ПО (строительные блоки средних и больших программных продуктов). Примеры программного кода приведены на языке С#. Помимо паттернов, т. е. как надо делать, приведены и антипаттерны, т. е. негативные практики разработки, что может значительно повысить качество кода читателя, при избегании последним подобных практик. Также в главе описывается важность оформления кода, чем обычно страдают новички в программировании. Также приводятся несколько мощных практик поддержки и модификации кода: рецензирование кода, рефакторинг кода, оптимизация кода.

В главе 8 приводятся классификация и описание различных видов тестирования программ. Изучение тестирования позволит читателю в значительной степени улучшить качество программы, автоматизировав тестирование продукта и тем са-

мым выйдя на уровень управления качеством ПО. Достижение подобного уровня позволит формально контролировать наличие и появление ошибок в разрабатываемом программном продукте, не к этому ли мы все стремимся?

Глава 9 описывает информационное обеспечение процесса разработки. Программирование в блокноте, перенос программного кода к коллеге на флеш-носителе и ручная сборка установочного файла, это, конечно, хорошо, но уже создано большое количество инструментов и определено большое количество позитивных практик, позволяющих в значительной степени избавиться от рутины, переложив её на компьютер, оставив разработчику, только творчество в виде написания кода. О подобных инструментах и подходах и рассказывает эта глава.

При написании этого пособия авторы надеялись создать простой и ёмкий ресурс, воспользовавшись которым, можно быстро влиться в сложный и интересный мир разработки ПО. Надеемся, что это у нас получилось!

## Соглашения, принятые в книге

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



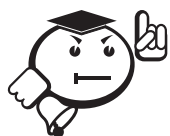
.....  
*Этот блок означает определение или новое понятие.*  
 .....



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



.....  
*Эта пиктограмма означает цитату.*  
 .....



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



..... **Пример** .....

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



.....  
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.  
.....



## Выводы

.....  
Эта пиктограмма означает выводы. Здесь автор подводит итоги, обобщает изложенный материал или проводит анализ.  
.....



## Контрольные вопросы по главе

.....



---

## Глава 1

# ПРОЦЕСС СОЗДАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

---

Сегодня практически нет людей, которые не знают, что такое компьютер. Помимо этого, достаточно распространённой тенденцией является наличие не одного, а нескольких компьютеров, которые могут быть представлены большим количеством различных электронных приборов, это и как различные мобильные устройства: ноутбуки, смартфоны, планшеты и пр., так и менее мобильные: персональные компьютеры, мейнфреймы, серверы и суперкомпьютеры. Компьютеры представляют собой сложные высокотехнологичные электронные устройства, способные выполнять большое количество действий за небольшой промежуток времени. Для корректной (и не очень в случае вирусов) работы компьютеров программистами пишется невообразимо огромное количество компьютерных программ на множестве различных языков программирования.



.....  
*Процесс создания компьютерных программ называется **программированием**.*  
.....

Программирование — это достаточно молодая динамически развивающаяся отрасль, которая год за годом прирастает новыми методиками программирования, сложными технологиями и различными мифами. Самого программиста люди, плохо знакомые с особенностью отрасли, представляют то шаманом, то волшебником, но все они сходятся в том, что написание программного кода, **СКОРЕЕ ВСЕГО**, есть очень запутанный и сложный процесс. В то же время, те, кто профессионально разрабатывает ПО, это **ЗНАЮТ**.

Сложность — не новинка в мире разработки ПО. Один из пионеров информатики Эдсгер Дейкстра обращал внимание на то, что компьютерные технологии — единственная отрасль, заставляющая человеческий разум охватывать диапа-

зон простирающийся от отдельных битов до нескольких сотен мегабайт информации, что соответствует отношению 1 к  $10^9$ , или разнице в девять порядков [1]. Такое гигантское отношение просто ошеломляет. Дейкстра выразил это так:



.....  
*«По сравнению с числом семантических уровней средняя математическая теория кажется почти плоской. Создавая потребность в глубоких концептуальных иерархиях, компьютерные технологии бросают нам абсолютно новый интеллектуальный вызов, не имеющих прецедентов в истории».*  
 .....

Разумеется, за прошедшее с 1989 г. время сложность ПО только выросла, и сегодня отношение Дейкстры вполне может характеризоваться 15 порядками.

Дейкстра пишет, что ни один человек не обладает интеллектом, способным вместить все детали современной компьютерной программы [2], поэтому нам — разработчикам ПО — не следует пытаться охватить всю программу сразу. Вместо этого мы должны попытаться организовать программы так, чтобы можно было безопасно работать с их отдельными фрагментами по очереди. Целью этого является минимизация объёма программы, о котором можно думать в конкретный момент времени. Можете считать это своеобразным умственным жонглированием: чем больше умственных шаров программа заставляет поддерживать в воздухе, тем выше вероятность того, что вы уроните один из них и допустите ошибку при проектировании и кодировании.

Всё перечисленное выше приводит нас к выводу, что любая программа должна создаваться с должной ответственностью и в процессе создания ПО разработчик должен учитывать большое количество разных, зачастую взаимоисключающих требований. За всё время создания программ сам процесс создания ПО претерпел незначительные изменения. Менялись технологии и инструменты разработки ПО, появлялись новые парадигмы программирования, однако основные этапы разработки ПО остались практически неизменными.

Прежде чем остановиться подробнее на этапах разработки ПО, необходимо остановиться на метафорах, которые используются при разработке. Важность использования метафор при разработке объясняется их простотой. Всегда проще ориентироваться на какую-то знакомую концепцию для понимания сложного процесса, чем не опираться ни на что.

## 1.1 Метафоры при создании ПО

Сама концепция и классификация метафор взята у автора бестселлера «Совершенный код» Стивена Макконнелла [3].

Нельзя недооценивать важность использования метафор для описания определённых процессов. Применение метафор для описания чего-либо позволяет нам абстрагироваться от определённых аспектов самого процесса, остановившись на значимых частях и описав с помощью наиболее простых и знакомых концепций. Многие открытия в науке были сделаны при помощи метафор.

Кинетическая теория газов была создана на основе модели «бильярдных шаров», согласно которой молекулы газа, подобно бильярдным шарам, имеют массу и совершают упругие соударения.

Волновая теория света была разработана преимущественно путём исследования сходств между светом и звуком. И свет, и звук имеют амплитуду (яркость — громкость), частоту (цвет — высота) и другие общие свойства. Сравнение волновых теорий звука и света оказалось столь продуктивным, что учёные потратили много сил, пытаясь обнаружить среду, которая распространяла бы свет, как воздух распространяет звук. Они даже дали этой среде название «эфир» — но так и не смогли её обнаружить. В данном случае метафора является примером, когда излишняя убедительность может привести к неверным посылкам.

Поиск хороших метафор — дело не очень простое. Ведь хорошая метафора должна быть простой, согласоваться с основными аспектами процесса, который она описывает. Также метафора должна обладать определённой теоретической целостностью, для того, чтобы без большого количества расширений описывать необходимый объект. Также, как мы уже выяснили, хорошая метафора не должна вводить использующих её людей в заблуждение.

Как и в случае совершения важных открытий хорошие метафоры могут помочь лучшему пониманию вопросов разработки ПО. Во время лекции по случаю получения премии Тьюринга в 1973 г., Чарльз Бахман упомянул переход от доминировавшего геоцентрического представления о Вселенной к гелиоцентрическому. Геоцентрическая модель Птолемея не вызывала почти никаких сомнений почти 1400 лет. Затем, в 1543 г. Коперник выдвинул гелиоцентрическую теорию, предположив, что центром Вселенной на самом деле является Солнце, а не Земля. В конечном итоге такое изменение умозрительных моделей привело к открытию новых планет, исключению Луны из списка планет и переосмыслению места человечества во Вселенной.

Переход от геоцентрического представления к гелиоцентрическому в астрономии Бахман сравнил с изменением, происходившим в программировании в начале 1970-х. В это время центральное место в обработке данных стали отводить не компьютерам, а базам данных. Бахман указал, что создатели ранних моделей стремились рассматривать все данные как последовательный поток карт, «протекающий» через компьютер (компьютеро-ориентированный подход). Суть изменения заключалась в отведении центрального места пулу данных, над которым компьютер выполняет некоторые действия (подход, ориентированный на БД).

Сегодня почти невозможно найти человека, считающего, что Солнце вращается вокруг Земли. Столь же трудно представить программиста, который бы думал, что все данные можно рассматривать как последовательный поток карт. После опровержения старых теорий трудно понять, как кто-то когда-то мог в них верить. Ещё удивительнее то, что приверженцам этих старых теорий новые теории казались такими же нелепыми, как нам старые.

Геоцентрическое представление о Вселенной мешало астрономам, которые сохранили верность этой теории после появления лучшей. Аналогично компьютеро-ориентированное представление о компьютерной вселенной тянуло назад исследователей компьютерных технологий, которые продолжали придерживаться его после появления теории, ориентированной на БД.

Разработка ПО — относительно молодая область науки. Она недостаточно зрелая, чтобы иметь набор стандартных метафор. Поэтому на данный момент нет метафоры, которая бы полностью описывала процесс разработки ПО, однако есть метафоры наиболее подходящие к разработке ПО. Их виды будут рассмотрены ниже.

## Популярные метафоры, характеризующие разработку ПО

Множество метафор, описывающих разработку ПО, смутит кого угодно. Дэвид Грайс утверждает, что написание ПО — это наука [4]. Дональд Кнут считает это искусством [5]. Уоттс Хамфри говорит, что это процесс [6]. Ф. Дж. Плождер и Кент Бек утверждают, что разработка ПО похожа на управление автомобилем, однако почти все приходят к почти противоположным выводам [7, 8]. Алистар Кокберн сравнивает разработку ПО с игрой [9], Эрик Реймонд — с базаром [10], Энди Хант и Дэйв Томас — с работой садовника, Пол Хекель — со съёмкой фильма «Белоснежка и семь гномов» [11]. Фред Брукс упоминает фермерство, охоту на оборотней и динозавров, завязших в смоляной яме [12]. Какие метафоры самые лучшие?

### **Литературная метафора: написание кода.**

Самая примитивная метафора берёт своё начало во фразе «написание кода». В этом случае разработка ПО представляется как написание литературного произведения. Программист представляется писателем, который строка за строкой пишет код и превращает его в работающую программу.

Эта метафора достаточно полно описывает разработку ПО в одиночку, однако разработка ПО в общем она описывает не очень адекватно. Разработка над современным программным продуктом редко ведётся в одиночку, что значительно усложняет восприятие этой метафоры. Как можно представить добавление нового текста в уже законченное произведение. При этом добавление текста может носить не только характер добавления «главы», но и добавление нескольких новых строк. Понятно, что в таком случае уже не получится «целого» литературного произведения, а в случае с ПО, такой вариант вполне возможен.

Помимо этого, литературная метафора подразумевает, что уже готовое произведение не изменяется, а печатается и далее остаётся неизменным. В случае разработки ПО большая часть работы может приходиться на период после разработки. Из общего объёма работы над типичной программой обычно две трети выполняется после выпуска первой версии программы, а очень часто эта цифра достигает 90% [13].

В литературе поощряется оригинальность идей, в то время как при разработке ПО такие подходы носят названия «изобретения квадратного колеса» и «создание своего велосипеда». При разработке ПО принято повторно использовать существующие компоненты для ускорения выпуска программного продукта.

К сожалению, литературная метафора была увековечена в одной из самых популярных книг по разработке ПО — книге Фреда Брукса «The Mythical Man-Month» («Мифический человеко-месяц») [12]. Брукс пишет: «Планируйте выбросить первый экземпляр программы: вам в любом случае придётся это сделать». Перед глазами невольно возникает образ мусорного ведра, полного черновиков.

Такой совет может быть полезным, если вы работаете над небольшим «произведением», однако, если вы работаете над крупным корпоративным приложением, разработка которого по сложности может сравниться со строительством небоскрёба или изготовлением космической ракеты, выбрасывать первый экземпляр программы может быть крайне затратным и неэффективным подходом. На такой подход можно было бы опираться, если бы вы обладали бесконечным количеством ресурсов (временных и человеческих), чего, естественно, не бывает.

Поэтому правильным выводом из этой метафоры являются слова Крейга Зеруни:



.....  
 «Если вы планируете выбросить первый экземпляр программы, вы выбросите и второй».

### **Метафора жемчужины: медленное приращение системы.**

Иногда говорят о приращении ПО. Приращение можно рассматривать как аналогичный процесс при формировании жемчужины. Приращение характеризует процесс формирования жемчужины за счёт отложения небольших объёмов карбоната кальция.

Эта метафора наводит на мысль о том, что программа должна создаваться итерационно, путём наращивания программного каркаса (песчинки в случае жемчужины) необходимым функционалом (небольшими объёмами карбоната кальция). Такое наращивание должно выполняться инкрементально.

При инкрементальной разработке сначала создаётся самая простая система, которую можно запустить. Такая система может не принимать реальные данные и даже не работать с ними, однако она изначально будет обладать необходимой программной архитектурой (крепким скелетом), которая в дальнейшем инкрементально будет обрастать нужным функционалом.

Эффективность такого подхода можно подтвердить следующими примерами. Фред Брукс, который в 1975 г. предлагал выбрасывать первый экземпляр ПО, заявил, что за десять лет, прошедших с написания знаменитой книги «Мифический человеко-месяц», ничто не изменило его работу и её эффективность так радикально, как инкрементальная разработка [12]. Также оценил инкрементальную разработку Том Гилб в революционной книге «Principles of Software Engineering Management» [14], в которой он представил метод эволюционной поставки программы (evolutionary delivery) и разработал многие основы современного гибкого программирования (agile development). Многие современные методологии также основаны на идее инкрементальной разработки ПО.

Основным достоинством представленной метафоры является то, что она достаточно просто расширяется в сторону правильных технологий разработки ПО и сложно поддаётся неуместному расширению.

### **Строительная метафора: построение ПО.**

Сравнение разработки ПО с процессом строительства является более полезным чем сравнения с «написанием» или с «выращиванием». Построение ПО подразумевает наличие стадий планирования, подготовки и выполнения, проработанность которых зависит от типа и сложности проекта.

Например, при построении собачьей будки не нужно большого планирования, т.к. мы используем немного ресурсов и в любой момент, как в случае с литературной метафорой, можем начать всё с начала и сделать новую будку. В сравнении с программой в несколько тысяч строк мы можем достаточно быстро изменить необходимую часть кода, а в худшем случае и переписать всю программу заново, при этом это не отнимет много времени и результат будет получен практически сразу.

При строительстве большого небоскрёба, как, например, Бурдж Кхалифа в ОАЭ, необходимо будет всё заранее спланировать, заранее просчитать все ресурсы, основательно продумать все вопросы логистики, откуда будет доставляться необходимый материал и хватит ли его, ведь в случае просчётов в планировании (недостаточный расчёт фундамента, отсутствие или нехватка ресурсов и пр.) при строительстве небоскрёба это может вылиться в миллионы долларов перерасхода и десятки лет совокупного простоя оборудования и рабочих. Абсолютно также должна происходить разработка крупных программных проектов. В случае обнаружения ошибок планирования при реализации ПО, совокупные расходы могут приблизиться к таким же расходам, как и при строительстве небоскрёба.



### Пример

Для детального описания аналогии дальше будет разобрано построение дома. Сперва необходимо решить, какой тип здания вы хотите построить, что аналогично определению проблемы при разработке ПО. Затем вы с архитектором должны разработать и утвердить общий план строительства, что похоже на разработку программной архитектуры. Далее вы чертите подробные чертежи и нанимаете бригаду строителей — это аналогично детальному проектированию ПО. Вы готовите стройплощадку, закладываете фундамент, создаёте каркас дома, обшиваете его, кроете крышу и проводите в дом все коммуникации — это похоже на конструирование ПО. Когда строительство почти завершено, в дело вступают ландшафтные дизайнеры, маляры и декораторы, делающие дом максимально удобным и привлекательным. Это напоминает оптимизацию/рефакторинг (рефакторинг — от англ. *refactoring* реорганизация кода) ПО. Наконец, на протяжении всего строительства вас посещают инспекторы, проверяющие стройплощадку, фундамент, электропроводку и всё, что можно проверить. Это идентично обзорам и инспекциям проекта.

И в программировании, и в строительстве увеличение сложности и масштаба проекта сопровождается ростом цены ошибок. При этом основные затраты будут связаны не с материалом, хотя он безусловно является дорогим, а с трудом рабочих. Материалы, необходимые для создания программного продукта, стоят дешевле, чем стройматериалы, однако затраты на рабочую силу в случае программирования будут более дорогими.

Ещё одной параллелью со строительством в программировании является использование уже готовых компонентов. Никому в голову не придёт самостоятельно разрабатывать и создавать холодильник, телевизор, стиральную машину, диван,

если это всё можно приобрести. Создавая программу, лучше руководствоваться теми же принципами. Если уже есть готовый программный компонент, которые делает нужные вам вещи, лучше использовать его, т. к. его разработчики уже прошли через все ошибки при разработке этого компонента, с которыми вам придётся столкнуться при самостоятельной разработке. Обычно невыгодно писать компоненты, если уже есть готовые.

Однако если вам нужна оригинальная мебель, бытовая техника с дополнительными характеристиками (низким энергопотреблением) — всё это, возможно, будет необходимо делать на заказ. Также и с компонентами ПО. Если вам необходим компонент с высокими требованиями к производительности, придётся вкладывать время на реализацию собственного компонента с необходимыми характеристиками. Иногда в разработке ПО встречаются такие задачи, что приходится переписывать низкоуровневые компоненты операционной системы, для того чтобы удовлетворять определённым требованиям.

И программирование, и строительство должны иметь этап тщательного планирования. Тщательное планирование не значит исчерпывающее или чрезмерное. Обычно планируются основные структурные компоненты, а при дальнейшей разработке определяется, как должны быть реализованы эти компоненты. Хорошо спланированный проект открывает большие возможности для модификации решения на поздних этапах работы. При наличии определённого опыта, вы можете опустить большое количество деталей, для того чтобы общий проект программы был достаточным, а недостаток планирования не привёл к дорогостоящим ошибкам.

Проблема изменения ПО приводит нас к ещё одной параллели. Перемещение несущей стены на 15 см обходится гораздо дороже, чем перемещение перегородки между комнатами. Аналогично внесение структурных изменений в архитектуру программы обходится гораздо дороже, чем изменение нескольких методов расчёта.



.....  
Аналогия со строительством позволяет нам лучше понять работу над большими программными проектами. Кейперс Джонс сообщает, что программная система из одного миллиона строк требует в среднем 69 видов документации [15]. Спецификация требований к такой системе обычно занимает 4000–5000 страниц, а проектная документация вполне может быть ещё в 2 или 3 раза более объёмной. Маловероятно, чтобы один человек мог понять весь проект такого масштаба или даже прочитать всю документацию, поэтому подобные проекты требуют более тщательной подготовки.  
.....

Метафора построения-конструирования может быть расширена во многих других направлениях, именно поэтому она столь эффективна. Благодаря этой метафоре отрасль разработки ПО обогатилась многими популярными терминами, такими как архитектура ПО, леса, конструирование и фундаментальные классы.



.....  
*Комбинирование метафор.* Не стоит думать, что одна метафора может полностью описать процесс разработки ПО, поэтому основным советом будет использовать комбинации метафор для работы над программными проектами. Использование метафор — дело тонкое. Чтобы метафора привела вас к ценным догадкам, её необходимо расширить. Однако нельзя допускать ситуации, когда метафора подвергается недопустимым расширениям, иначе это может привести вас к поиску эфира.  
 .....

## 1.2 Этапы разработки ПО

Разработка ПО, как уже было сказано ранее — это сложный многоитерационный процесс. Однако в этом процессе можно выделить несколько основных этапов, через который проходит в той или иной степени практически каждый программный продукт. Итак, основными этапами разработки ПО являются:

- появление задачи;
- составление ТЗ и анализ задачи;
- составление проекта программной системы;
- реализация проекта системы;
- тестирование ПО;
- поддержка ПО.

Все перечисленные этапы могут соблюдаться как со стопроцентной точностью, так и с определёнными изменениями. Изменения этапов могут быть связаны с особенностями работы проектной команды, масштабом проекта и имеющимися на разработку ресурсами. При прочтении следующих глав читатель может подробно познакомиться с описанием каждого этапа разработки ПО и используемыми на каждом этапе современными технологическими решениями. Ниже вкратце приведено описание каждого этапа.

### Появление задачи

У читателя может возникнуть закономерный вопрос при прочтении этой главы: «Каким же образом влияет появление задачи на разработку ПО?». Следует сказать, что формально появление задачи никаким образом не относится к разработке ПО, однако без задачи, которую необходимо будет решить, коллективу разработчиков нечем будет заняться и, как следствие, они останутся без заработной платы.

Появление задачи является очень важным этапом с той точки зрения, что любая система (программная или любая другая) направлена на решение определённой задачи. Правильное определение самой задачи для решения может быть основополагающим фактором будущей успешности или неуспешности программного продукта.



В учебном пособии не будут приведены современные техники целеполагания и поиска успешных идей, однако следует помнить, что разрабатываемая программа должна после окончания решать поставленную на самом первом этапе задачу, иначе, как несложно догадаться, это может вылиться в нецелевое расходование ресурсов проекта и несданный в срок проект.

Пару слов следует сказать об источнике появления задачи. Разработка программного продукта может быть инициирована несколькими возможными способами: снаружи программного коллектива внешним заказчиком или внутри программного коллектива для решения насущной проблемы, например создания внутренней информационной системы, или для реализации новой перспективной идеи. Помимо этого, внешний способ появления задачи может быть также разделён на задачи по реализации нового программного продукта или задачи по поддержке существующего ПО. Все три типа появления задачи обладают своими уникальными особенностями, о которых следует задуматься ещё до решения о «вступлении в войну».

## Составление ТЗ и его анализ

После появления задачи каждый разработчик сталкивается с таким этапом разработки, как составление технического задания (сокращённо ТЗ). Составление корректного ТЗ уже прочно вошло в профессиональный жаргон разработчиков программ по большей части из-за сложности в коммуникации между командой программистов и заказчиком. Иногда ТЗ даже расшифровывают как Точка Зрения, подчёркивая, что она может различаться у заказчика и разработчиков. Это приводит к тому, что разработчикам раз за разом приходится добавлять что-то в программу, изначально не готовую к внесению подобной функциональности. Такое «доведение» программы на последних этапах обычно выливается в нарушение архитектурной целостности продукта, что приводит к большому количеству ошибок на этапе тестирования.



.....  
Поэтому к процессу составления ТЗ необходимо подходить очень тщательно и формально настолько, насколько позволяет заказчик разрабатываемого ПО. Все изначальные требования должны быть отражены на бумаге и заверены обеими сторонами: заказчиком или его представителем и представителем команды разработчиков.  
.....

Также следует помнить, что составление ТЗ не должно быть независимым от заказчиков, очень важно на всех этапах разработки привлекать заказчика к работе над проектом, иначе, из-за различных Точек Зрения на проект, на выходе может получиться продукт, который коренным образом не устроит заказчика. А это может означать несколько вариантов: либо заказчик не оплатит работу, т. к. полученный продукт не будет отвечать необходимым требованиям, либо будет необходимо в срочном порядке переделывать программу, что приведёт к переутомлению членов команды разработки и понижению их мотивации и, как следствие, к падению производительности.

Существует определённая группа разработчиков, которая полагает, что ТЗ должно составляться заказчиком, следует заметить, что это наиболее желаемый вариант развития событий, т. к. все мечтают о готовом и грамотно составленном ТЗ. Однако реалии более жестоки, чем этого хотелось бы, обычно заказчик вообще не понимает, что такое ТЗ и как его нужно делать, поэтому заранее нужно закладывать время на донесение до заказчика важности ТЗ и принципов его составления.



.....  
 Ещё раз: *грамотно составленное ТЗ — это результат плотной работы двух сторон!*  
 .....

Если читателю попадётся такой заказчик, который самостоятельно принесёт грамотно составленное ТЗ, — обязательно напишите авторам этого пособия, т. к. подобное событие является равновероятным встрече со снежным человеком!

При разработке ПО грамотно составленное ТЗ играет роль фундамента, если обратиться к строительной метафоре, описанной ранее. Все перечисленные в ТЗ требования являются формальными сваями, на которых дальше будет стоять ваш дом — разрабатываемый программный продукт. Представьте себе, что внезапно возникнет необходимость добавить пару новых свай, чтобы надстроить ещё несколько этажей многоэтажного дома. Такая работа является не только крайне сложной, но и, иногда, вовсе невозможной из-за определённых конструктивных особенностей здания. Так и грамотно составленное ТЗ. Если на этапе составления ТЗ упустить или неправильно описать часть требований, к концу разработки программы затраты на исправление первоначальной ошибки могут значительно превысить все мыслимые и немыслимые пределы. Для того, чтобы не быть голословным, в таблице 1.1 приведена средняя стоимость исправления дефектов в зависимости от времени их внесения и обнаружения [3].

Таблица 1.1 – Средняя стоимость исправления дефектов в зависимости от времени их внесения и обнаружения

Время обнаружения дефекта					
Время внесения дефекта	Выработка требований	Проектирование архитектуры	Конструирование	Тестирование системы	После выпуска ПО
Выработка требований	1	3	5–10	10	10–100
Проектирование архитектуры	—	1	10	15	25–100
Конструирование	—	—	1	10	10–25

Эти данные говорят о том, что если на этапе выработки требований и составления ТЗ будет найдена ошибка и её исправление составит 5000 рублей, то после вы-

пуска ПО исправление такой ошибки может обойтись в 50 000, а то и в 500 000 рублей. Согласитесь, что лучше подойти с необходимым усердием к каждому этапу разработки ПО, чтобы общие затраты на разработку не превысили заложенные ресурсы. Также следует отметить, что в большинстве проектов основная часть усилий на исправление дефектов всё ещё приходится на правую часть таблицы 1.1, а значит, на отладку и переделывание работы уходит около 50% времени типичного цикла разработки ПО [15–23]. В десятках компаний было обнаружено, что политика раннего исправления дефектов может в два и более раз снизить финансовые и временные затраты на разработку ПО [24]. Это достаточно веский довод в пользу как можно более раннего нахождения и решения проблем, особенно на этапе составления ТЗ.

Подробное описание ТЗ, его состава и роли в разработке ПО будет представлено в главе 2.

После формального заверения ТЗ обеими сторонами, а это следует сделать обязательно, дабы исключить возможности неконтролируемого изменения требований к продукту, требования к ПО необходимо проанализировать и решить, каким образом будет строиться дальнейший процесс разработки. Этот этап формально называется «Анализом технического задания».

При анализе технического задания необходимо распределить роли в команде (о ролях подробнее будет описано в главе 3), выбрать инструменты (подробное описание в главах 5–9) и методологию разработки (подробное описание в главе 4) для реализации продукта. Подробное описание всех этих процессов будет представлено далее, однако следует сказать, что выбор правильного инструментария значительно отразится на скорости реализации программы. Лучше всего необходимость выбора правильного инструментария описывает пословица «стрелять из пушки по воробьям».

За счёт чего может значительно сократиться время разработки? Современный программный мир не стоит на месте, а эволюционирует с невероятной скоростью. Поэтому прежде чем браться за написание компонент будущей программной системы, не помешало бы проверить, а может быть, именно ваш «велосипед» уже кем-то был написан. Может быть он хорошо подойдёт вашей задаче и заодно сэкономит время и средства на разработку собственных программных компонент. Вопрос выбора существующих программных компонент или написания собственных является намного более глубоким и, даже, более философским, чем может показаться на первый взгляд. В данном учебном пособии этот вопрос не будет подробно освещаться, поэтому лучше ознакомиться с реальным опытом разработчиков, описанным во всемирной паутине.

Помимо этого, очень много времени может быть сэкономлено с помощью выбора правильной программной методологии. Например, возможно, изначально вам будет комфортно разрабатывать в процедурном стиле, увеличивая программный код только добавлением новых процедур, однако когда таких процедур станет более 100, ваша команда начнёт путаться в них, тратить большее время на чтение кода и, скорее всего, допускать ошибки. Разделение системы по объектно-ориентированной методологии изначально позволит вам избежать «пробуксовки».

Результатом анализа ТЗ будет перечень инструментов, которые позволят эффективно разрабатывать необходимый продукт. Следует заметить, что выбранные

на этом этапе инструменты, не следует воспринимать, как монолит, который нельзя ни за что менять. Может так оказаться, что в середине процесса разработки будет найдена более перспективная технология или методология. В таком случае следует рассчитать стоимость внесения изменений в процесс разработки (переписывание существующего кода, его тестирование и отладку) и выгоду от принятого решения. В случае значительного перевеса естественно следует начать использовать другие инструменты. На заключительных этапах разработки, следует крайне осторожно относиться к подобным идеям.

## Проект системы

После анализа задания, в котором выбираются инструменты разработки, технологии и методологии программирования, идёт этап проектирования программной системы. Результатом этого этапа будет комплект проектной документации, называемый проектом системы. Как уже было показано выше, не стоит относиться к этому этапу разработки спустя рукава.



.....  
*Проект программной системы — это документ, описывающий будущую программную систему, взаимодействие её частей, ответственность каждой части системы таким образом, чтобы можно было приступить к реализации программы без значительных модификаций документа.*  
.....

Следует заметить, что существуют некоторые документы, регламентирующие состав проекта системы, например [25]. Разработка проектной документации с такой высокой степенью формализации необходимо лишь в том случае, если в разработку вовлечено большое количество людей и разрабатывается достаточно сложный программный продукт.

Перед написанием проекта системы следует определиться с основными понятиями и инструментами, которые будут использованы. Например, обычно, для представления архитектуры программы выбирают тот или иной вид диаграмм. Сейчас наибольшее распространение получил графический язык, называемый Unified Modelling Language (UML) [26], который позволяет представить различные аспекты функционирования программной системы с помощью определённых стандартом UML условно-графических обозначений. Сам UML будет подробно рассмотрен в главе 6. Однако до выбора UML следует удостовериться, что все разработчики компетентны в использовании выбранного языка моделирования и владеют им в нужной степени для чтения проекта системы.

Обычно небольшому проекту системы достаточно описать несколько аспектов с помощью диаграмм — это элементарные единицы функциональности программы — высокоуровневую и детализированную архитектуру программы. Для каждого из них существует свой тип UML-диаграммы. Для описания элементарных единиц функциональности программы и связей между ними применяют диаграмму вариантов использования (Use Case Diagram), для описания высокоуровневой архитектуры используют диаграмму пакетов (Package Diagram), для описания детализи-

рованной архитектуры используется диаграмма классов (Class Diagram). Все эти диаграммы описаны в стандарте UML 2.0. В языке моделирования присутствует ещё много типов диаграмм, которые описывают другие, более сложные аспекты поведения программной системы, эти диаграммы будут вкратце описаны в главе про UML.

При проектировании высокоуровневой и детальной архитектуры программы следует руководствоваться принципом неизбыточности, т. е. в проект системы следует включать только те диаграммы и только с такой степенью проработки, которой действительно заслуживает проектируемая часть системы. Очень часто, под влиянием перфекционизма и желания «подстелить солому» на будущее можно разработать такой проект системы, который называется ёмким англоязычным термином *over-engineering*. Реализация такого проекта будет трудоёмкой и долгой, а заложенные по всему проекту точки расширения системы, могут помешать быстрому наращиванию функционала и приведут не только к сложности восприятия программы, но и к большему количеству ошибок.

Иногда менеджмент в больших корпорациях не понимает, зачем нужен этап проектирования программных систем. Обычно менеджер заходит в комнату, где работают программисты, и бывает очень недоволен тем, что не видит программного кода на мониторах. В программистском жаргоне это явление даже получило своё название WISCA или WIMP: «Why Isn't Sam Coding Anything?» (Почему Сэм не пишет код?) или «Why Isn't Mary Programming?» (Почему Мэри не программирует?).



.....  
Для того чтобы избежать подобных проблем, необходимо заранее объяснить начальству необходимость разработки проекта системы, при этом можно сослаться на исследования, объединённые в таблице 1.1.  
.....

Помимо различных диаграмм, описывающих систему, в проект системы очень часто включают макеты пользовательского интерфейса. На этих макетах схематически, очень часто карандашом на листке бумаги, рисуются основные элементы пользовательского интерфейса. Проектирование пользовательского интерфейса достаточно деликатный вопрос, откладывание решения которого может надолго продлить разработку проекта. Обычно заказчик очень щепетильно относится к разработке пользовательского интерфейса, т. к. именно интерфейс видит тот, для кого разработана программная система. Следует учесть, что при необходимости тщательной проработки пользовательского интерфейса, следует перенести его макеты в техническое задание и заранее утвердить их с заказчиком — утверждённый пользовательский интерфейс не позволит заказчику в дальнейшем вносить коррективы и усложнять уже сделанный проект системы. В настоящем пособии в главе 5 будут вкратце рассмотрены этапы проектирования пользовательского интерфейса программы, а также задачи, которые стоят перед разработчиком пользовательского интерфейса.

Имея на руках готовый проект системы, включающий основные диаграммы, и утверждённый проект системы, можно смело приступать непосредственно к разработке программы, т. е. к этапу программирования.

## Реализация

Программирование или реализация является именно тем этапом, к которому обычно не терпится преступить, минуя все остальные этапы. Так обычно поступают ещё неопытные студенты программистских специальностей. Вместо выработки у себя дисциплины и аналитического мышления при разработке проекта системы, которые необходимы для решения сложных системных задач, такие студенты стремятся сразу «ввязаться в войну», решая локальные задачи с помощью выбранного языка программирования. Такой подход оправдывает себя лишь при сравнительно небольшой системной сложности задачи, однако даже в такой задаче проект системы не будет лишним, поэтому если этапу проектирования уделено значительное внимание и сделан грамотный проект системы, то реализация превращается в решение небольших программистских задач.

Реализация программы обычно включает два основных этапа:

- высокоуровневое кодирование;
- детализированное кодирование.



.....  
**Высокоуровневое кодирование** — это реализация с помощью программного кода основной архитектуры программы — скелета программной системы.

**Детализированное кодирование** — это наращивание функционала — мышцы программы.

.....

Реализация любой программной системы — сложная системная задача, которая требует развитых аналитических способностей человеческого мозга. При этом следует учитывать, что по последним исследованиям среднестатистический человеческий мозг может одновременно удерживать не более  $7 \pm 2$  предмета. Именно из-за этой особенности человеческого мышления всё развитие технологий программирования было направлено на уменьшение сложности разработки и поддержки программных систем, Стив Макконелл даже называет управление сложностью — Главным Техническим Императивом Разработки ПО [3]. Для решения задачи по управлению сложностью программной системы была использована концепция декомпозиции сложной задачи на подзадачи, когда, разделив задачу на более мелкие части, мы решаем сперва эту небольшую задачу, не используя мозг для хранения информации о всей задаче. Решив эту задачу, мы можем спокойно приступить к решению другой задачи и так, пока не сможем решить общую задачу. Этот же принцип используется при разработке ПО: сперва сложная система разбивается на подсистемы, определяются особенности высокоуровневой архитектуры, а потом уже реализуются все составные части программы, с применением детального кодирования.

Для решения задачи построения высокоуровневой архитектуры программы сейчас принято использовать так называемые шаблоны проектирования — это несколько высокоуровневых программных каркасов, используемых для определённого рода функций. Шаблоны были предложены в 1970 году, как сборник пра-

вил программирования определённых типов программ или программных модулей. Шаблоны также направлены на управление сложностью программной системы, т. к. они предоставляют разработчику уже готовый набор программных конструкций для заранее определённых типов задач. Про шаблоны более подробно будет рассказано в главе 7.

После разработки с помощью шаблонов высокоуровневой архитектуры программисты приступают к детальному кодированию, т. е. наполнению необходимых классов функциональностью. Следует отметить, что на протяжении всего методического пособия будет использоваться именно объектно-ориентированная парадигма программирования. Объектно-ориентированная методология с основной единицей программной системы — классом выбрана как наиболее используемая на сегодняшний день парадигма.

После создания классов системы необходимо позаботиться о контроле изменения программы, т. е. упростить себе дальнейшую работу с помощью современных средств программирования. Такими средствами на самом низком уровне программы служат юнит-тесты (иногда их называют блочными или модульными тестами). На тему юнит-тестирования написано много книг и предложена даже методология разработки ПО через тестирование (Test Driven Development). В общем случае тесты служат для контроля общедоступных (интерфейсных) частей разработанных классов. Сейчас достаточно сложно понять, как что такое интерфейсная часть класса, так и что такое юнит-тест, поэтому в данной главе эти определения не будут подробно рассматриваться. Информация о юнит-тестах приведена в главе 9.

## Тестирование

Этап тестирования ПО — это финальный этап, на котором можно исправить имеющиеся в программе дефекты. Под тестированием понимается очень широкий спектр действий: от «прокликивания» пользовательского интерфейса, до разворачивания системы непрерывной разработки с автоматизацией процесса тестирования и поднятия его на качественно новый уровень — управление качеством ПО.

Тестирование ПО относительно новая и непрерывно развивающаяся отрасль разработки программных систем. Разработаны уже большое количество техник, технологий и фреймворков (от англ. *framework* — каркас, программная платформа) для тестирования. Написано большое количество книг. Сейчас уже становится общепринятой практикой открытие в командах разработки ПО отделов контроля качества. Учебное пособие не ставит целью подробное изучение современных тенденций тестирования ПО, однако краткий обзор последних достижений в тестировании представлен в главе 8.

## Этап поддержки

После передачи готового продукта заказчику наступает этап поддержки программного продукта. Под поддержкой программного продукта имеется в виду исправление появляющихся в ПО ошибок, которые не были выявлены на этапе тестирования.

Фиксация необходимых для решения задач может осуществляться одним из перечисленных способов:

- отправка отчёта об ошибке с помощью заранее встроенной системы обратной связи в разработанной программе. Разработка такой системы очень сложная системная задача, т.к. для создания подобной системы необходимо заранее проработать систему логирования (с англ. *log* — системный журнал), которая будет сохранять все возникающие ошибки в файл определённого формата; систему передачи сообщений с такими файлами команде разработчиков через интернет; автоматическую генерацию задачи по поступающим от пользователей сообщениям;
- предоставление пользователю электронного адреса или телефона отдела поддержки ПО, куда он может обратиться напрямую;
- открытие пользователям системы управления проектов, в которую они могут вносить появившиеся ошибки или пожелания в виде задач — тикетов (слэнг от английского *ticket* — задача, электронный запрос). Подробнее о системах управления проектами будет рассказано в главе 9.

Обычно общение с пользователями ПО не входит в обязанности разработчиков, поэтому программисты общаются с ними опосредовано — в форме решения задач в системе управления проектами. Для работы над возникающими у пользователей вопросами создаётся отдел поддержки, который сегодня в сленговой форме все называют саппортом (от англ. *support* — поддержка). Работа отделов сопровождения уже обросла специфическим юмором, например первым советом при возникновении какой-то проблемы является: «А вы пробовали выключить и включить заново компьютер/программу/веб-сайт?», — на что технически грамотные люди очень часто обижаются.



## Контрольные вопросы по главе 1

1. В чём заключается проблема сложности при разработке ПО?
2. Зачем нужны метафоры для разработки ПО и какие метафоры вы знаете?
3. В чём суть строительной метафоры разработки ПО?
4. Для чего нужен проект системы?
5. В чём заключаются явления WISCA и WIMP?



---

## Глава 2

# ТЕХНИЧЕСКОЕ ЗАДАНИЕ

---

Одной из самых главных ошибок многих разработчиков является небрежное отношение к ведению проектной документации, в частности к техническому заданию. Вечная и излишняя самоуверенность начинающего программиста приводит к мысли: «Зачем тратить время на бумажки, если и так всё понятно? Программист должен писать код!». С этой мысли и начинается заблуждение, которое в лучшем случае приведет к затягиванию сроков разработки и падению качества продукта, в худшем — потере заказчика, клиентов и денег.

Программист должен писать код. Но прежде, чем начать писать код, нужно убедиться, что программист правильно понимает задачу. Как раз для этого и необходимо техническое задание.

Есть различные стандарты оформления технического задания [27, 28], однако, если от вас не требуют обратного, вы можете вести данный документ так, как считаете нужным. Главное, чтобы данный документ отвечал на следующие вопросы:

1. Каковы цель и назначение разрабатываемой программы?
2. Какие задачи должна решать программа для достижения указанной цели?
3. Какова исходная проблема, решением которой должна являться программа?
4. Каков контекст использования данной программы? Или в другом варианте: кто конечный пользователь продукта и как он будет его использовать?
5. Каким критериям качества должен отвечать продукт?
6. Каким дополнительным требованиям (например, системным) должен удовлетворять продукт?

В зависимости от заказчика, в документе также изначально могут присутствовать такие пункты, как:

1. Прототип пользовательского интерфейса.
2. Полное описание функциональности программы (бизнес-логика).
3. Этапы разработки/приёмки продукта заказчиком.
4. Временные, финансовые и человеческие ресурсы.

Если данных пунктов нет в документе изначально, то их стоит добавлять в процессе их обсуждения и разработки. При этом как и первоначальный вариант документа, так и все его изменения и добавление новых пунктов должны утверждаться обеими сторонами, например подписью ответственных с обеих сторон. Отвечая на перечисленные выше вопросы, техническое задание позволит добиться следующих результатов:

- *Единое понимание конечного продукта у заказчика и разработчика.* Это очень важно, так как если по завершению разработки заказчик увидит продукт не таким, как он себе представлял, он откажется от оплаты трудов программистов. Потерянное время и невозмещенные расходы. Лучше, если вы изначально расставите все точки над «i».
- *Решение конфликтов с заказчиком на этапе приёмки.* Во время приёмки готового продукта заказчиком могут возникнуть конфликты. В частности, заказчик может утверждать, что вы не сделали некоторые функции, о которых он вас просил, либо вообще сделали всю программу не так, как он себе представлял. В такой момент очень хорошо иметь под рукой техническое задание, подписанное заказчиком. Фактически, в техническом задании должны быть перечислены все функциональные возможности и критерии качества продукта. И в таком случае техническое задание будет играть роль своего рода чек-листа: если все пункты технического задания выполнены, то и претензий со стороны заказчика быть не может — сам не доглядел, когда подписывал техническое задание. Если же продукт не соответствует техническому заданию, то претензии заказчика обоснованы, и вам придется исправлять. Таким образом, оформленное и вовремя подписанное техническое задание может разрешить спорные моменты. Естественно, что если в процессе разработки одна из сторон определила, что отдельные пункты технического задания нереализуемы или требуют пересмотра, обе стороны должны внести все необходимые изменения в ТЗ прежде чем продолжать работу.
- *Единое понимание конечного продукта среди разработчиков.* Если разработка ведется в команде (а два человека уже считаются командой), то также важно добиться того, чтобы все участники проекта понимали, что и для чего они разрабатывают. В конечном счете, неправильное толкование задач продукта может привести к созданию неправильного (неудобного для выполнения исходных задач) пользовательского интерфейса и к неправильной архитектуре приложения.
- *Определение процесса разработки.* Техническое задание позволяет определить многие ключевые моменты: объём работы, количество разработчиков и их роли, методологию разработки, определение этапа тестирования и т. д.

Здесь перечислены только основные причины необходимости технического задания, на самом деле их гораздо больше.



.....  
 Но главное, что необходимо понять, техническое задание — это лучшее средство защиты *ваших* интересов перед заказчиком.  
 .....

Несмотря на то, что техническое задание навязывает вам сроки выполнения и ограничивает вас, на самом деле, для разработчиков техническое задание становится незаменимым руководством и доказательством правильности выполнения работы. Поэтому отнеситесь к его составлению и утверждению с должным вниманием и не забывайте включать в него любые изменения и дополнения. Важно отметить, что *любое изменение* технического задания *должно быть утверждено* заказчиком и вами.



.....  
Если у вас или заказчика нет возможности физически подписать изменения технического задания (что, кстати, может происходить довольно часто, чем может надоесть заказчику), подтверждения в виде ответного электронного письма от заказчика будет вполне достаточно. Но такое подтверждающее письмо должно быть, это в ваших же интересах! При этом никогда не стесняйтесь обсуждать с заказчиком техническое задание и сам проект. Как уже говорилось, чем лучше вы будете понимать друг друга, тем лучше будет для продукта в целом.  
.....

## 2.1 Составление технического задания

В предыдущем пункте были перечислены основные вопросы, которые должно отражать техническое задание. Важно понять, что это не просто перечень некоторых формальных пунктов, а идентификация самого продукта, и каждый следующий пункт является следствием предыдущего. Таким образом, само техническое задание должно быть связанным, логичным документом. Рассмотрим каждый из них более подробно.

### Исходная проблема

Любая разработка должна решать некоторую проблему. Если ваш продукт не решает никакой проблемы, то он обречен, несмотря на его качество и функциональные возможности. Отметим, что разрабатываемый продукт должен решать некоторую проблему конечного пользователя — того человека, который будет непосредственно использовать вашу программу. Если продукт не решает проблем пользователя, пользователь не приобретет продукт.

Проблему нужно сформулировать максимально четко, уложившись в одно предложение. При этом проблема должна быть только одна. Решение нескольких проблем одновременно может привести к неправильной расстановке акцентов при проектировании программы и её пользовательского интерфейса. В итоге, программа не сможет хорошо решать ни одну из поставленных проблем.



## Пример

### Пример постановки проблемы

*У студентов, обучающихся по направлениям программирования, нет удобных программных средств для рисования блок-схем в соответствии с требованиями ЕСПД.*

Проверить качество данной формулировки можно по принципу бритвы Оккама — попробуйте удалить из данного предложения хотя бы одну из его частей. Если удалить часть «в соответствии с требованиями ЕСПД», то проблема сразу теряет свою актуальность, так как на проверку в мире существует много программ как для рисования, так и для рисования блок-схем в частности. А вот программ, учитывающих требования ЕСПД, фактически нет.

Если же удалить из предложения часть «у студентов, обучающихся по направлениям программирования», то сразу становится непонятным, кто же является конечным пользователем и у какой именно группы людей возникла данная проблема.

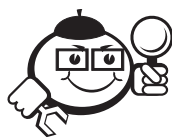
При формулировке проблемы отслеживайте следующие признаки:

- Актуальность проблемы должна легко проверяться.
- Должна присутствовать конкретизация целевой аудитории, так как проблема возникает у конкретных людей, а не на пустом месте.
- Проблема должна быть одна.

После формулировки очень важно задуматься, а действительно ли это является проблемой для пользователя. В случае нашего примера, а действительно ли на практике студентам требуется данная программа? Может таких программных продуктов не существует, потому что в них нет необходимости? Неправильный ответ на данный вопрос приведет к бессмысленной и никому ненужной разработке продукта.

### Цель и задачи

После формулирования проблемы необходимо определить цель и задачи. Так же, как и проблема, цель должна быть строго одна, и её формулировка должна точно решать исходную проблему. Задач может быть много, они определяют некоторые количественные и качественные признаки, выполнение которых будет сигнализировать о достижении поставленной цели.



## Пример

Цель — разработать быструю программу для создания небольших блок-схем согласно требованиям ЕСПД.

Задачи:

- Программа должна предоставлять рисование графическими примитивами, соответствующими требованиям ЕСПД.
- Программа должна предоставлять возможность работы с блок-схемами до 50 элементов.
- Программа должна иметь возможность сохранения блок-схем в распространенные графические форматы.
- Программа должна иметь возможность редактирования ранее созданных блок-схем.

Как видно, цель напрямую связана с ранее поставленной проблемой, и достижение поставленной цели решит исходную проблему. Но цель не является простым копированием исходной проблемы, здесь появляются несколько важных характеристик конечного продукта. Обратите внимание на фразы «быструю программу» и «небольших блок-схем» — эти фразы описывают ключевые моменты готовой программы, так как первоначальной целевой аудитории нет нужды в сложных программных комплексах, работающих с многостраничными диаграммами. Студентам нужна быстрая простая программа для рисования блок-схем уровня учебных лабораторных работ. Как же понять, что разработанная программа действительно быстрая, и что она работает с небольшими блок-схемами? Эти понятия мы конкретизируем в задачах.

В первую очередь мы указываем, что всё, что создает пользователь, должно соответствовать ЕСПД — ведь это основное предназначение программы.

Далее мы уточняем, а что же является небольшой блок-схемой. Очевидно, что студенты в своих работах не будут создавать блок-схемы, включающие более 50 элементов. Именно это число мы закрепляем в задачах.

Также важно понять, что данные диаграммы также предназначены для формирования студенческих отчетов, а значит, данные блок-схемы должны легко переноситься в другие текстовые и графические редакторы. Для этого укажем задачу о совместимости с распространенными графическими форматами.

Ну и конечно же, у пользователя должна быть возможность редактирования ранее созданных схем для исправления ошибок или изменения описанных алгоритмов.

Заметьте, что выполнение каждой из задач можно будет легко проверить, имея готовый продукт. Выполнение же всех задач означает достижение цели, а значит, программа готова для решения проблем пользователей.

## Контекст использования и конечный пользователь

Одним из определяющих факторов при проектировании программы и пользовательского интерфейса является контекст использования и конечный пользователь. Вам нужно четко определить, кто является вашей целевой аудиторией и в каких условиях он будет использовать ваш продукт.

Программные продукты можно разделить на *продукты широкого применения* и *узкоспециализированные*. Если ваше приложение относится к первой группе, то в дальнейшем всё взаимодействие с программой вы должны сделать максимально простым и удобным для любого пользователя. Доступностью приложения также обуславливается и выбор платформы разработки — устройства, операционной системы, языка программирования. В случае узкоспециализированных приложений в своих требованиях вы учитываете знания и навыки вашей целевой аудитории, что позволяет создавать более сложные элементы управления и инструменты взаимодействия.

Специализированность приложения может определяться:

- *возрастной группой* — определите возрастной интервал вашей целевой группы;
- *полом* — если ваше приложение ориентировано на конкретный пол, то вашу целевую аудиторию можно сократить вдвое. Разумеется, здесь есть свои плюсы и минусы;
- *профессией* — знание конкретной предметной области пользователем в значительной степени влияет как на функциональность программы, так и на интерфейс;
- *социальным статусом* — да, в нашем современном мире есть и такое деление, и данный фактор нужно учитывать при проектировании приложения. Пример, продукция компании Apple позиционирует себя как продукция высокого качества, близкая к совершенству. Как следствие, добрая часть их аудитории — люди, желающие ощутить свою привилегированность или уникальность по отношению к другим. Они выделяют себя в иную социальную группу. Другую же часть потребительского рынка такое позиционирование продукции, наоборот, отпугивает, из-за чего потенциальная целевая аудитория значительно уменьшается;
- *условиями использования* или *контекстом* — те или иные продукты пользователь может использовать в различных ситуациях и различных условиях. Примером могут послужить смартфоны и ноутбуки. Оба данных продукта ориентированы в первую очередь на молодую аудиторию, не конкретизированы на пол, профессию или социальный статус. Однако контекст использования у данных продуктов различный — разговаривать по телефону пользователь, как правило, будет на ходу, например, по дороге на работу. Возможно, у такого пользователя одна из рук даже будет занята сумкой, портфелем, документами и тому подобным. Следовательно, продукт должен быть удобен при ходьбе и даже использовании одной рукой. Ноутбук, в свою очередь, предназначен для более серьезной работы, его использование предполагает наличие стола или хотя бы сидячего места. Такие детали контекста напрямую повлияли на внешний вид данных устройств и способы взаимодействия с ними;
- *специальными ограничениями* — среди потенциальных пользователей всегда присутствует группа людей с ограниченными возможностями. Это могут быть ограничения по зрению, слуху и т. д. Если в вашей целевой аудитории присутствует данная группа, постарайтесь сделать программу более комфортной и для неё.

## Критерии качества

Следующим шагом является определение качественных и количественных признаков, согласно которым мы будем отслеживать качество продукта в целом. Данный пункт напрямую вытекает из задач, однако является более развернутым и может учитывать требования, определенные после выяснения целевой аудитории. Как и задачи, критерии качества должны быть легко определяемы, поэтому отдавайте предпочтение количественным признакам — таким признакам, которые можно легко посчитать. Если продукт не соответствует некоторому определенному вами количественному уровню, вероятно, его необходимо доработать и улучшить.

Необходимость определения критериев качества позволяет конкретизировать этап тестирования продукта. В частности, каждый придуманный вами критерий качества должен обладать приоритетом. В дальнейшем, при разработке и тестировании можно перераспределить ресурсы согласно данным приоритетам. Приоритеты, разумеется, расставляются из соображений важности для конечного пользователя.

## Обзор аналогов

Обязательный пункт решения любой технической проблемы. Всегда есть возможность, что уже существует готовое решение вашей проблемы, которое полностью решает поставленные задачи, таким образом можно сэкономить время и деньги на разработку.

Суть обзора аналога — поиск всех возможных продуктов, решающих исходную или схожую задачу и дальнейшее их сравнение согласно определенным критериям качества. Многие разработчики упускают вторую часть обзора — сравнение по выбранным критериям. Ведь вам необходимо определить, действительно ли нет решения исходной задачи. Если хотя бы один из аналогов удовлетворяет большинству критериев качества — вероятно, нет необходимости в разработке собственного решения. Если же ни один из аналогов не удовлетворяет ключевым критериям — в разработке вашего продукта есть какой-то смысл.

## Дополнительные требования

В данном пункте уточняются различные технические детали и прочие нюансы. Например, требуемая платформа разработки или описание системных требований, языки локализации, способы распространения продукта, особенности технической документации пользователя и тому подобное. Фактически, содержание данного пункта во многом определяется самой программой. Иногда дополнительные требования распределяются по остальным разделам ТЗ.

## Описание функциональных возможностей программы

Ключевой раздел технического задания, в котором описывается вся бизнес-логика приложения. Именно данный раздел является перечнем работ для разработчиков, и именно для этого раздела необходимо внимательное исследование и продумывание всех предыдущих пунктов.

Если задачи и критерии качества описывают, что программа должна делать, то в данном разделе описывается *как и какими инструментами* программа должна решать поставленную задачу.



## Пример

*Задача:* программа должна предоставлять рисование графическими примитивами, соответствующими требованиям ЕСПД.

Функциональные возможности:

1. Интерфейс должен предоставлять панель инструментов с перечислением всех возможных примитивов, представленных в виде пиктограмм.
2. Выбор определенной пиктограммы приводит к переключению режима рисования мышью в соответствии с выбранным примитивом.
3. Режим рисования мышью для примитива «Стрелка» — пользователь наводит на рабочее пространство диаграммы мышью в позицию начала стрелки. Необходимо кликнуть мышью для размещения начала стрелки. После размещения начала стрелки пользователь должен навести мышью в позицию конца стрелки и кликнуть мышью еще раз. До выполнения размещения конца стрелки на рабочем пространстве должна отображаться предполагаемая стрелка. До выполнения размещения конца стрелки пользователь может нажать клавишу Esc для прерывания отрисовки стрелки. В таком случае, сбрасывается размещение начала стрелки и пользователь может начать рисование стрелки с другой начальной позиции. Началом стрелки считается направление «Откуда передается управление», концом стрелки считается направление «Куда передается управление»...

Заметьте, для решения одной задачи, сформулированной в единственном приложении, потребовалось несколько абзацев для описания того, как программа должна работать для решения данной задачи. Также обратите внимание, что в примере содержится неполное описание только одного примитива. Полное описание функциональности для решения данной задачи потребовало бы несколько десятков страниц технического задания.

Такое описание может показаться избыточным, однако помните, что одна из проблем, решаемых ТЗ, это устранение двусмысленностей и неясностей в работе будущей программы. Описание в первую очередь должно быть *достаточным* для понимания работы функциональности.

## Прототип пользовательского интерфейса

Этап прототипирования предназначен для уточнения технического задания, в ходе которого создается макет приложения, обсуждающийся разработчиками, заказчиками и фокус-группой. Также данный этап позволяет спроектировать пользо-



вательский интерфейс приложения. Более подробно прототипирование рассматривается в главе 5.

## План разработки проекта и ресурсы

Данный пункт завершает техническое задание. Все перечисленные выше разделы полностью описывают готовый продукт, в последнем разделе перечисляются шаги и этапы разработки. Фактически, если продукт достаточно большой (разработка займет более полугода), логичнее предоставлять программу заказчику по частям по мере готовности. Такой подход позволит получить отзыв от заказчика и внести корректировки в техническое задание при необходимости. Соответственно, в ТЗ описывается количество подобных этапов, сроки их выполнения, функциональные возможности, реализуемые на каждом этапе, стоимость выполнения каждого этапа и другие необходимые ресурсы.



## Контрольные вопросы по главе 2

1. Какова основная цель составления технического задания?
2. Какие пункты должны быть отражены в техническом задании?
3. Чем отличается цель от задачи?
4. В чем особенность определения целевой аудитории будущего продукта?
5. Чем обуславливается необходимость поддержки и актуализации технического задания?

---

## Глава 3

# КОМАНДНЫЕ РОЛИ

---

Разработка программного обеспечения — командная работа. Конечно, на первых этапах, очень часто, разработка может вестись одним человеком-энтузиастом, но в этом случае ему придется решать очень много проблем, начиная с проектирования архитектуры и заканчивая маркетингом (в случае коммерческого приложения). Поэтому программы, разрабатываемые одним человеком, очень редко становятся коммерчески успешными: одному человеку очень сложно выполнить все требуемые работы на высоком уровне. Отсюда следует, что для создания качественного продукта необходима команда. Важнейшее преимущество команды — разделение обязанностей, что позволяет ускорить процесс разработки и сделать его максимально эффективным за счет того, что каждый делает то, что умеет делать лучше всего.

Можно дать формальное определение.



.....  
**Команда** — это группа индивидов, которые распределяют между собой рабочие операции и ответственность за получение конкретных результатов.  
.....



.....  
Три важных характеристики команды:

- **Взаимозависимость** — каждый член команды вносит свой индивидуальный вклад в общую работу. Другие члены команды зависят от работы каждого. В команде все делятся рабочей информацией друг с другом. Члены команды являются равноправными участниками процесса деятельности и имеют возможность влиять друг на друга.

- Разделяемая ответственность — ответственность за командные цели понимается и разделяется всеми.
- Результат — ответственность за командные результаты разделяется всеми членами группы и фокусирует групповую активность.

.....

Пожалуй, следует отметить, что не каждая группа людей является командой, потому что команда нацелена на результат. Можно привести пример — несколько студентов собрались вместе и решили написать игру (очень много студентов-программистов на 2–3 курсе университета этим увлекаются). Эта группа не является командой: часто студенты этим занимаются для общения, для повышения своих навыков программирования или дизайна, часто цель таких мероприятий — «пощупать» процесс разработки. То есть, грубо говоря, разработка ради самого процесса разработки. Получится ли что-нибудь в конце — не так важно, именно поэтому большинство студенческих разработок, по сути, бесконечные и не имеют никаких результатов. Деятельность же членов команды направлена в первую очередь на достижение цели, а уже затем на общение и тому подобное.

Создание команды — дело сложное и трудно формализуемое. Однако существуют множество научных работ, которые анализируют различные команды и выделяют рекомендации по их формированию. В этих работах выделяются две роли членов команды — функциональная (проектная) и командная. Функциональные роли относятся к должностным обязанностям и охватывают навыки и умения, знания и опыт. Командные роли отражают способ, с помощью которого мы выполняем свою работу. Командная роль также определяется нашими врожденными и приобретенными личными качествами. Далее в этой главе будут рассмотрены эти роли.

### 3.1 Командные роли по Белбину

Концепция командных ролей Белбина [29] развивалась в соответствии с представлениями о том, что эффективная работа команды возможна тогда, когда команда сбалансирована с точки зрения представленности членами группы необходимых командных ролей. Эта модель очень популярна в мире.

Командная роль — это определенная тенденция работы в команде. Это, во-первых, вклад человека в общее дело, во-вторых, определенное поведение, и, в-третьих, взаимодействие с другими членами команды. Доктор Белбин выделил всего восемь командных ролей: генератор идей, исследователь ресурсов, координатор, организатор, аналитик, душа команды, практик, контролер. Эти роли, или тенденции, у каждого человека проявляются по-разному. Одни из них присущи генетически либо очень хорошо развились на протяжении жизни. Это сильные роли. Используя их, человек достигает наилучших результатов. Есть роли, они называются развиваемыми, которые не являются самыми предпочтительными для человека, однако вполне способны проявляться в некоторых ситуациях и могут быть

развиты со временем. И наконец, есть роли слабые, те роли, в которых человек не будет успешным и которые необходимо делегировать.

*Генератор идей (Plant)* — изобретателен, обладает богатым воображением, умеет решать нестандартные проблемы. Отличительные черты характера: индивидуалист, часто серьезно мыслящий, открыт к восприятию новых идей, склонный к творчеству и нововведениям, чаще интроверт, дистанцированный в личном общении. Развитое воображение, высокий уровень интеллекта, креативность. С трудом акцентирует внимание на практических деталях, протоколе. Склонен отстраняться, погружаясь в размышления, предпочитает работать в одиночку. Предпочитает относительную свободу.

*Исследователь ресурсов (Resource investigator)* — энтузиаст, общителен. Исследует возможности, устанавливает контакты с нужными людьми, выявляет новые возможности, способен реагировать на возникающие трудные задачи. Экстротверт, любопытен, коммуникабельный, нуждается в свободе действий. Склонен терять интерес, как только первоначальное очарование проходит, может быть слишком оптимистичен и не критичен. Исследователя стимулирует наличие людей, команды рядом. Нормально реагирует на кризис и оказываемое давление.

*Координатор (Chairman)* — умеет четко формулировать цели, продвигать решения, делегировать полномочия. Часто является более зрелой личностью. Социальный лидер, умеет слушать, хорошо говорит. Для него характерны: спокойствие, уверенность в своих силах, контроль и самообладание. Часто имеет ясное представление о сильных и слабых сторонах всех участников команды, принимает их в соответствии с их достоинствами и без предубеждений поощряет вклад всех, кто потенциально способен улучшить работу команды. Не всегда наделен выдающимися интеллектуальными и творческими способностями. Предпочитает использовать имеющиеся ресурсы, направляя их на достижение командной задачи. Внимание сосредоточено на команде.

*Организатор (Shaper)* — изобретателен, обладает богатым воображением, человек с идеями. Динамичен, бывает неуживчив, но есть способности и стремления преодолеть инерцию и неэффективность, бывает самодоволен. Легко реагирует на провокации, раздражителен и нетерпелив, склонен травмировать чувства других людей. Как и координатор является сильным лидером, но для решения определенной задачи. Может конфликтовать с координатором из-за стилей руководства.

*Аналитик (Monitor Evaluator)* — проныцателен, осмотрителен, расчетлив, обладает стратегическим мышлением. Волевой тип личности. Видит все альтернативы. Объективен при анализе проблем и оценке идей, все взвешивает, по своей природе — инспектор. Рассудителен, неэмоционален, предусмотрителен. Не всегда умеет мотивировать людей, воодушевлять, но умеет анализировать мысли других людей, никогда не делает скоропалительных выводов.

*Душа команды (Team worker)* — дипломатичен, восприимчив. Умеет слушать, предотвращает трения между членами команды. Социально ориентирован, достаточно мягок, чувствителен. Не всегда способен принимать решения в моменты кризиса, но хорошо адаптируется к изменениям. Действует для достижения гармонии и поддержания духа команды. Умеет работать под руководством «трудных» людей (организаторов).

*Практик (Company worker)* — дисциплинированный, надежный, консервативный, эффективный. Умеет реализовывать идеи в практических действиях. Обязателен и предсказуем. Обладает хорошими организаторскими способностями, практическим здравым смыслом, трудолюбив. Акцентирует свое внимание на графиках, планах мероприятия. Выявляет лучший способ достижения результата. Часто берет на себя задания, которые другие не хотят выполнять.

*Контролер (Completer/Finisher)* — старателен и добросовестен. Ищет ошибки и упущения. Контролирует сроки выполнения поручений. Способен выполнять свои обещания, стремится все выполнить на высоком уровне. Иногда склонен волноваться без особых причин, не стремится делегировать полномочия и обязанности. Стремится выполнить задание должным образом, концентрируясь на деталях. Больше обеспокоен результатом, нежели способом его достижения. Спокойно относится к контролю и приемлет большинство типов руководителей.

В последнее время к этим ролям добавляют еще одну.

*Специалист* — профессионал в узкой области знаний. Самостоятельно мыслящий и организующий свою работу, приверженный своему направлению. Обладает редкими знаниями и навыками. Может не видеть общую картину. Приверженец высоких стандартов. Не любит пристального контроля, особенно со стороны тех, кто обладает меньшей по сравнению с ними компетенцией.

Команде нужны исполнители большинства ролей. Но принцип комплектации команд Белбина не требует полного 100%-го перекрытия всех ролей отдельными членами группы. По его мнению, идеальное количество человек в команде равняется шести. Это означает тот факт, что каждый член команды берет на себя функции больше чем одной роли. Относительная значимость каждой из ролей зависит от специфических требований ситуации. Главным для построения эффективной команды является нахождение и поддержание необходимого баланса ролей для конкретной ситуации.

С точки зрения модели Белбина, к командной работе способен не каждый человек. Способных к этому людей около 70%. Они могут иметь 2–3 сильные командные роли, между которыми они имеют некоторый выбор, и 2–3 роли, к которым они не приспособлены полностью. Оставшиеся 30% людей проявляют себя более эффективно при индивидуальной работе.

Комплектация управленческих команд как правило осуществляется на базе уже сложившегося «костяка» команды. И основная задача сводится к доукомплектованию команды совместимыми людьми либо выявлению существенных ролевых конфликтов, противоречий и принятию действия для уменьшения ролевой неопределенности.

## 3.2 Функциональные роли

Как уже упоминалось, функциональные роли относятся к должностным обязанностям и охватывают навыки и умения, знания и опыт членов команды. Каждый проект разработки ПО имеет свою организационную структуру, которая определяет ответственности и полномочия участников проекта. Чем меньше проект, тем больше ролей необходимо совмещать одному исполнителю.

Условно можно выделить пять групп участников проекта разработки ПР:

- Анализ. Постановка, документирование и сопровождение требований к продукту.
- Управление. Управление производственными процессами.
- Производство. Проектирование и разработка ПО.
- Тестирование. Тестирование ПО.
- Обеспечение. Производство дополнительных продуктов и услуг.

Ниже будут рассмотрены роли, относящиеся к каждой из групп.

Группа анализа состоит из следующих ролей:

- Бизнес-аналитик. Обеспечивает двустороннюю взаимосвязь между предметными экспертами заказчика и специалистами исполнителя путем сбора требований, их обработки, документирования и передачи специалистам исполнителя, а также путем доведения полученных результатов до представителей заказчика.
- Бизнес-архитектор. Разрабатывает бизнес-концепцию системы. Определяет общее видение продукта, его интерфейсы, поведение и ограничения.
- Системный аналитик. Отвечает за перевод требований к продукту в функциональные требования к ПО.
- Специалист по требованиям. Документирование и сопровождение требований к продукту.
- Менеджер продукта. Представляет в проекте интересы пользователей продукта.

Группа управления состоит из следующих ролей:

- Руководитель проекта. Руководитель проектной команды, ответственный за управление проектом, достижение целей проекта в рамках бюджета, в срок и с заданным уровнем качества.
- Куратор проекта. Как правило, руководитель высшего звена, который курирует проект, обеспечивает общий контроль и поддержку проекта финансовыми, материальными, человеческими и другими ресурсами. Куратор проекта отвечает за достижение проектом конечных целей и реализацию выгод для организации.
- Системный архитектор. Разрабатывает техническую концепцию системы. Принимает ключевые проектные решения относительно внутреннего устройства программной системы, её технических интерфейсов, взаимосвязь со смежными системами.

В производственную группу входят:

- Проектировщик. Занимается проектированием компонентов и подсистем в соответствии с общей архитектурой, разработкой архитектурно значимых модулей.
- Проектировщик базы данных.
- Проектировщик интерфейса пользователя.

- Разработчик. Занимается проектированием, реализацией и отладкой отдельных модулей системы.

Группа тестирования в проекте состоит из следующих ролей:

- Руководитель группы тестирования. Определяет цели и стратегии тестирования, управляет тестированием.
- Проектировщик тестов. Разрабатывает тестовые сценарии.
- Разработчик автоматизированных тестов.
- Тестировщик. Занимается тестированием продукта, анализом и документированием результатов.

К группе обеспечения можно отнести следующие проектные роли:

- Технический писатель.
- Переводчик.
- Дизайнер графического интерфейса.
- Разработчик учебных курсов, тренер.
- Участник рецензирования.
- Продажи и маркетинг.
- Системный администратор.
- Технолог.
- Специалист по инструментальным средствам.

Как уже говорилось, один участник проекта может совмещать несколько ролей, однако есть роли, которые лучше не совмещать, например разработчик и тестировщик.

Стоит отметить одну интересную вещь, не стоит нагружать программистов несвойственной им работой. Дело в том, что у программистов, достаточно специфическое мышление, которое не подходит для некоторых задач, таких как бизнес-анализ, проектирование эргономики, разработка документации. Эти задачи не имеют ничего общего с программированием и требуют совершенно иной квалификации. Зачастую программисту бывает сложно взглянуть на свою программу глазами пользователя, так сказать, со стороны, для него важнее видеть, как работает его программа внутри. Именно поэтому необходимо привлекать в проектную команду бизнес-аналитиков, эргономистов, художников-дизайнеров, технических писателей. Разделение труда и специализация позволяет более эффективно налаживать процесс разработки.



### Контрольные вопросы по главе 3

1. Для чего необходимо разделение ролей?
2. Что такое команда?
3. В чем разница между командной и функциональной ролью?
4. На какие группы можно разделить роли по Белбину?
5. Какие группы участников проекта можно выделить, какие у них функции?

---

## Глава 4

# МЕТОДОЛОГИИ РАЗРАБОТКИ ПО

---

В данной главе рассмотрены существующие методологии разработки ПО. Приведена их эволюция и представлены достоинства и недостатки каждой из них. Важность методологий разработки нельзя недооценивать при работе в команде.

### 4.1 Что такое методология разработки ПО и зачем она нужна?

Для того, чтобы понять, что такое методология разработки ПО, необходимо понимать, что такое методология в общем смысле.



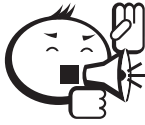
.....  
*Методология* — учение о методах, методиках, способах и средствах познания.  
.....

Исторически методологию разделяют на теоретическую и практическую. В нашем случае важным является именно практическое понимание методологии, как учение о методах, методиках, способах и средствах, направленных на решение практических проблем и целенаправленное преобразование мира. Под решением практических проблем и целенаправленным преобразованием мира мы будем понимать изучение процесса написания ПО, направленного на решение поставленных задач.



.....  
Другими словами, *методология разработки ПО* — это учение о методах, методиках, способах и средствах разработки ПО.  
.....





.....

Читатель может спросить: а зачем нужна ваша методология, если я могу разрабатывать программы и без всяких методологий, быстро и качественно! Автор может ответить только, что в таком случае методология не нужна читателю, потому что методология разработки ПО необходима для работы над общим проектом *не одного человека, а команды программистов.*

.....

Может показаться, что работа одного человека никак не отличается от работы команды программистов. Что самое интересное, так может казаться только тогда, когда ни разу не работал в команде. Для того, чтобы объяснить сложность работы в команде, следует разобраться, как происходит работа над проектом в команде.

Как мы выяснили в предыдущих главах, программирование в команде — это последовательное выполнение работы по разработке (программированию) модулей разрабатываемой системы. Зачастую оказывается, что разные модули и целые фреймворки пишут разные члены команды. И, как мы все знаем, очень часто не уделяют большого внимания разработке программной документации. Тем более, что наиболее удобный способ изучения программного модуля — обратиться к тому программисту, который его разрабатывал, если он находится в пределах досягаемости, и задать ему все интересующие вопросы. Представим такой вид разработки, как передачу сообщений между, в самом простом случае, двумя членами программной команды (рис. 4.1).

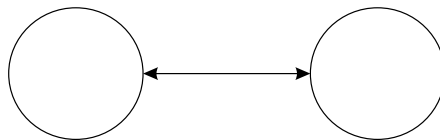


Рис. 4.1 – Схема коммуникаций в команде из 2-х человек

Такое представление расширяется и на способы организации работы команды с тремя и более членами (рис. 4.2).

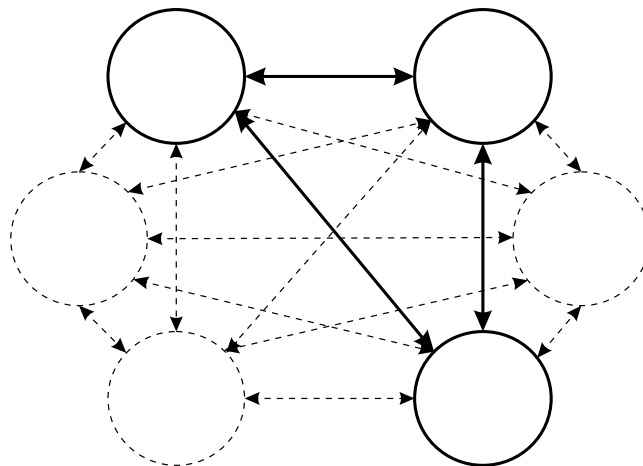


Рис. 4.2 – Схема коммуникаций в команде из 3-х и более человек

Как мы видим, сложность коммуникации растёт по экспоненте, что при большой команде может привести к тому, что большую часть времени разработчики будут уделять коммуникации со своими коллегами, но никак не работе над проектом. Для того чтобы показать продуктивность команды разработчиков разного количественного состава, ниже будет приведен график зависимости количества разработчиков от времени выполнения той или иной задачи [12] (рис. 4.3).

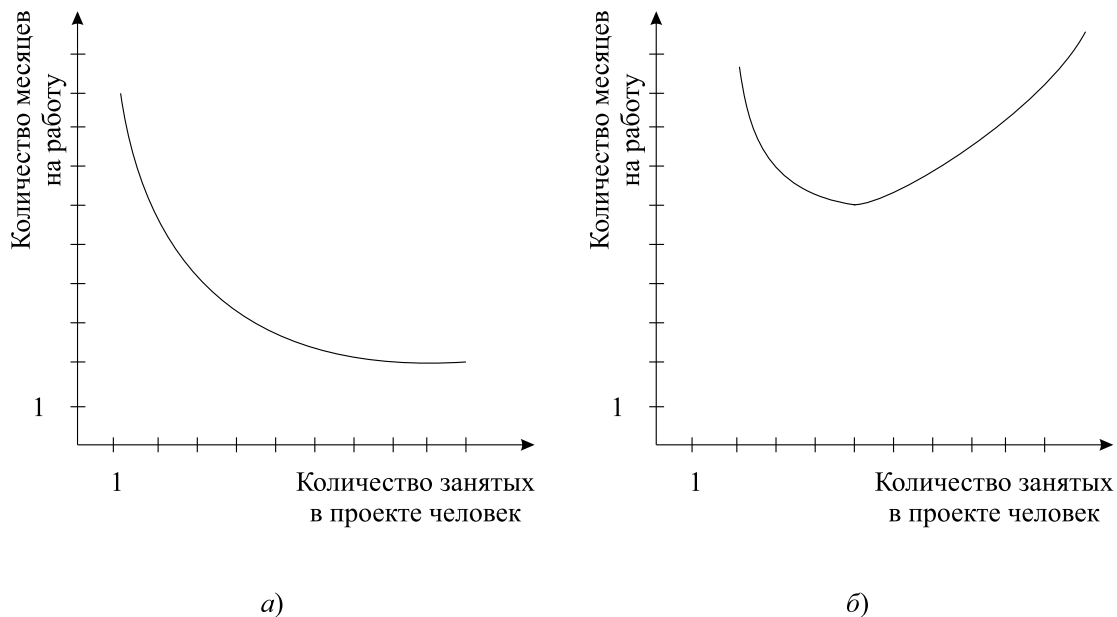


Рис. 4.3 – Зависимость времени от числа занятых: а) разделимая задача, требующая обмена данными; б) задача со сложными взаимосвязями

Как можно увидеть — если требуются сложные коммуникации между разработчиками, то увеличение количества разработчиков может привести к значительному увеличению сроков на разработку продукта. Можно подумать, что единственной реальной возможностью закончить проект в срок является разработка в команде не больше 4-х человек. Однако, руководствуясь этим доводом, довольно сложно было бы представить написание таких сложных систем, как операционные, или программ, решающих сложные системные задачи, например систем автоматизированного проектирования.



.....  
 При необходимости разработки сложной программной системы на помощь разработчикам приходит правильная организация труда, спланированная в соответствии с современными методологиями разработки ПО.  
 .....

## 4.2 Используемые методологии ПО

В этой главе приведены некоторые методологии разработки ПО, которые по мнению авторов являются наиболее распространёнными сегодня. Следует отме-

тить, что информация, представляемая в этой главе, не претендует на полноту описания, т. к. методологии всё время модернизируются и рождаются новые, поэтому на момент изучения учебного пособия вполне возможно, что в моде будут уже другие способы работы в команде. Однако принципы, которые сегодня используются при разработке ПО в существующих методологиях, скорее всего получат продолжение и в следующих модификациях, и материал, представленный ниже, не потеряет своей актуальности.

### 4.2.1 Водопадная методология

Первая из представленных методологий — водопадная методология. Первой она является по старшинству возникновения в сфере разработки ПО. Своё название она получила в русскоязычной литературе за счёт английского термина «waterfall methodology». Очень часто в отечественных источниках также встречается название «каскадная методология» (или модель) разработки ПО.

Первое упоминание водопадной методологии можно найти в статье доктора Винстона Ройса «Управление разработкой больших программных систем» в 1970 году [30]. В 70-х годах уже начали разрабатывать сложные программные системы для научных и оборонных нужд, нужд авиастроения и космической отрасли. Естественной реакцией на возникающие потребности в формализованном управлении процессом разработки ПО явилось появление каскадной методологии разработки ПО.

Можно подумать, что нет необходимости разговаривать о такой, достаточно старой методологии, однако читатель может удивиться, но на сегодняшний день очень много сложных программных систем до сих пор разрабатываются с помощью водопадной методологии. В качестве примеров использования водопадной методологии можно привести любую разработку, где заказчиком выступают государственные структуры. Почему водопадная методология в этом случае лучше всего подходит, будет рассказано после описания принципов работы методологии.

#### **Особенности методологии.**

Главной особенностью каскадной методологии является поэтапное выполнение всех шагов разработки ПО в строго фиксированном порядке, то есть переход от этапа к этапу осуществляется сверху вниз как *по водопаду*, откуда и получила название методология. Только завершение предыдущего этапа может служить началом следующего. Этап формирования требований также фиксируется в документах в качестве ТЗ и не изменяется в течение всего процесса разработки системы. Тут можно обрадоваться и сказать: «Да это же сказка!!! Требования, которые не изменяются!!!» Но эта радость может очень быстро испариться при первых финансовых отчётах после выпуска разрабатываемого продукта. Информация об этом будет приведена в описании недостатков методологии. Каждый этап разработки строго и формально описывается с помощью определённого вида документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

В соответствии с водопадной методологией разработку ПО можно разбить на следующие этапы: формирование требований; проектирование; реализация; тестирование; внедрение; эксплуатация и сопровождение.

Как видно, этапы методологии полностью повторяют этапы разработки, представленные в главе 1, поэтому подробное назначение каждого из этапов можно посмотреть там.

Преимущества использования каскадной методологии:

- 1) полная и согласованная документация на каждом этапе, которая позволяет безболезненно вводить в разработку новых участников. Помимо этого, большое положительное влияние на проект оказывают изначально сформулированные и неизменяемые требования;
- 2) легко определяются сроки и затраты на проект. Фактически на этапе планирования мы определяем объём работы на каждом из этапов проекта, делим этот объём на производительность команды, руководствуясь знаниями, приведёнными ранее на рисунке 4.3, и получаем общие сроки и затраты на проект.

Как показывает практика, все вышеперечисленные достоинства легко превращаются в недостатки при неумелом подходе менеджмента или определённой сложности разрабатываемого проекта. За время существования каскадной методологии она много раз подвергалась критике за чрезмерный формализм и негибкий подход к разработке. Поэтому ниже представлены основные недостатки такой методологии:

- 1) в водопадной методологии переход от одного этапа к другому может быть осуществлён только при полной корректности результатов предыдущего этапа. Мы знаем, что мир разработки очень сложен и невероятно изменчив, поэтому зачастую очень сложно предсказать, будет ли решена поставленная задача за приемлемые сроки и конечное (желательно наименьшее) число ресурсов проекта. Как следствие, затягивание любого из этапов проекта увеличивает сроки и бюджет разработки всей системы;
- 2) как уже было сказано в главе 1, невозможно всё предусмотреть заранее. Некоторые разработчики говорят: «Приступая к работе — составьте план, работайте, а на следующий день выкиньте свой план и составьте новый!»! Очень часто разработчикам приходится решать задачи, которые изначально не были заложены в проект. Это может произойти из-за нестабильной или недостаточно быстрой работы стороннего компонента, что приводит к написанию своего «велосипеда». Это может произойти просто из-за человеческого фактора, когда уйдёт один из разработчиков, а на введение в проект другого придётся потратить несколько незапланированных недель, а то и месяцев;
- 3) ошибки на каждом этапе разработки изначально являются критичными — это связано с длительностью разработки проекта. Следует понимать, что заказчик увидит работоспособную систему только на последних этапах разработки, поэтому любое неверно понятое требование или ошибка в архитектуре или функциональности будет найдена только после завершения работы над системой. Длительность разработки по водопадной методологии может варьироваться от полугода до бесконечности, соответственно по завершению разработки мы рискуем получить не тот проект, который задумывал получить заказчик, а это может вылиться в незапланированную модернизацию системы или выбрасывание всего проекта, как ненужного;

- 4) при работе на конкурентном поле по водопадной методологии мы не можем гарантировать гибкое изменение требований и функционала программы, что изначально снижает конкурентоспособность разрабатываемого продукта.

Обобщить всё вышесказанное поможет следующая статистика: по данным The CHAOS Manifesto, The Standish Group, 2012 — с 2002 по 2010 гг. 14% проектов были выполнены в срок и без превышения бюджетов с необходимым качеством, 57% их превысили и 29% проектов, реализующихся по водопадной методологии, были провалены [31].



.....

*Место для оптимизма.*

И тут читатель может заранее разувериться в представленной методологии и отказаться использовать её в принципе, однако обратимся к мысли, приведённой в начале главы. До сегодняшнего дня для разработки больших систем для государственных нужд до сих пор используют каскадную методологию разработки ПО. Это связано с особенностями работы государственных структур:

1. Долгосрочность планирования — позволяет планировать разработку тех или иных проектов на длительное время, что является одной из сильных сторон водопадной методологии.
2. Формализованный подход к документации — очень поощряется методологией и является одним из основополагающих её принципов.
3. Низкая конкурентоспособность — государственные структуры обычно существуют в рамках долгосрочных планов государства, а не в жёстких рамках рыночной экономики, что позволяет использовать для разработки каскадную методологию.

Помимо использования водопадной методологии в государственном секторе также существуют истории успеха, которые показывают её использование в коммерческих проектах, но не в чистом виде, а с элементами гибких методологий разработки, о которых речь пойдёт ниже.

.....

### 4.2.2 Гибкие методологии

В настоящем разделе будут представлены так называемые гибкие методологии разработки ПО (agile methodologies). Эти методологии появились как ответ закоренелому и неповоротливому каскадному подходу. Все гибкие методологии предполагают итерационную разработку с итерациями не более одного месяца. Гибкость таких методологий обусловлена как раз их итеративностью. За короткое время продукт проходит все фазы разработки: от постановки ТЗ до ввода в эксплуатацию,

что приводит к оперативной обратной связи от заказчика или пользователей и, как следствие, быстрому устранению текущих проблем продукта и добавлению новой функциональности.

Все принципы гибкой разработки ПО были сформулированы в феврале 2001 года, когда 17 разработчиков собрались для того, чтобы обсудить лучшие практики в разработке ПО. Главным аргументом в пользу нового подхода к разработке была мысль о том, что разработка программных систем должна укладываться в необходимые сроки и ресурсы, а для этого необходимо иметь более совершенные подходы к разработке. Новый подход был сформулирован в четырёх основных принципах гибкой разработки [32]:

- Люди и взаимодействие важнее процессов и инструментов.
- Работающий продукт важнее исчерпывающей документации.
- Сотрудничество с заказчиком важнее согласования условий контракта.
- Готовность к изменениям важнее следования первоначальному плану.

Следует сказать, что гибкие итерационные подходы к разработке позволили значительно сократить сроки разработки больших программных продуктов и снизить издержки на создание таких продуктов. Гибкие подходы получили большое распространение и на сегодняшний день стали стандартом де факто в сфере разработки ПО. При устройстве на работу положительную роль играет знание подобных методологий, т. к. отпадает необходимость в обучении новичка принципам работы в соответствии с тем или иным подходом.

Ниже будут представлены несколько основных методологий гибкой разработки ПО. Существующие подходы к разработке всё время модифицируются, поэтому помимо некоторых основных принципов методологии каждая команда добавляет в процесс некоторые индивидуальные черты, подгоняя общепризнанную методологию под свои конкретные потребности.

### **Scrum.**

Scrum является одной из наиболее применяемых сегодня гибких методологий разработки ПО. Чтобы понять, что такое Scrum, обратимся к англоязычному термину. Термин Scrum пришёл к нам из регби, в котором он обозначает борьбу за мяч в условиях постоянно меняющегося состояния на поле, т. к. игрокам мешают другие игроки и приходится на ходу придумывать обходные пути, искать другие траектории продвижения и применять различные уловки, чтобы доставить мяч на сторону соперника.

Руководствуясь подобной метафорой строится разработка проекта по данной методологии. Подход впервые был представлен Хиротака Такэут и Икудзиро Нонако в статье *The New New Product Development Game* [33]. Они отметили, что проекты, над которыми работают небольшие команды из специалистов различного профиля, обычно систематически производят лучшие результаты, и объяснили это как «подход регби». Далее подход совершенствовался и брался за основу при разработке в других программных компаниях, и в 2001 году Кен Швабер и Майк Бидл выпустили книгу «*Agile Software Development with SCRUM*» [34], в которой детально описали особенности подхода.

Общее описание методологии приведено на рисунке 4.4.

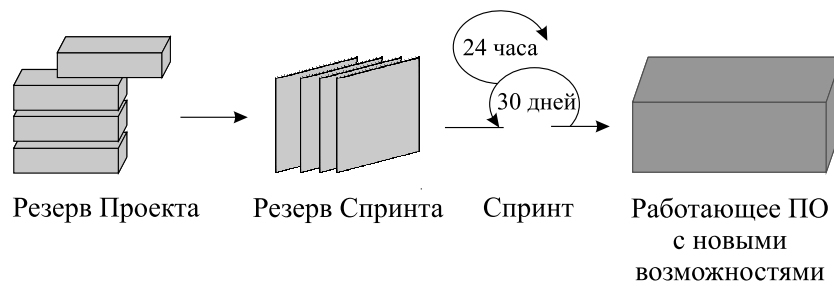


Рис. 4.4 – Схематическое обозначение работы по методологии Scrum

Для того, чтобы разобраться с тем, как работать по методологии, необходимо сначала разобрать имеющиеся термины.



.....  
**Резерв Проекта (Project Backlog)** — это список требований к функциональности разрабатываемой системы, упорядоченный по степени их важности.  
 .....

Все запланированные функции должны быть реализованы в процессе работы над проектом. Элементы этого списка называются «пожеланиями пользователя» (user story) или элементами резерва (backlog items). Резерв проекта всегда может редактироваться всеми участниками процесса разработки.



.....  
**Резерв Спринта (Spring Backlog)** — содержит функциональность, выбранную владельцем проекта из резерва проекта.  
 .....

О владельце проекта будет подробно рассказано при перечислении ролей в Scrum-команде. Все функции должны быть разбиты по задачам, каждая из которых оценивается командой. Каждый день команда оценивает объём работы, который нужно проделать для завершения спринта.



.....  
**Спринт** — итерация в Scrum, в ходе которой создаётся функциональный рост ПО.  
 .....

Спринт жёстко фиксирован по времени и проводится итерационно по времени в 2 или 4 недели. В отдельных случаях, к примеру, согласно Scrum-стандарту Nokia, длительность спринта должна быть не более 6 недель. Считается, что чем короче спринт, тем более гибким становится процесс разработки, позволяющие «обогать» возникающие на пути трудности, получая каждый раз обратную связь от владельца проекта за сравнительно небольшое время. Не следует принимать небольшую длительность спринта, как руководство к действию, т. к. каждый проект и каждая команда имеет собственные особенности и выбор длительности сприн-

та должен выполняться скорее экспериментальным путём, нежели какими-то внешними соображениями.

В качестве инструментов в Scrum-методологии используется так называемый burndown chart, к сожалению, этому инструменту нет корректного, по мнению автора, перевода на русский язык, поэтому в рамках данной главы будет применён термин «диаграмма выполнения задач». Диаграмма выполнения задач служит для наглядной демонстрации процесса выполнения имеющихся на спринт задач. Обычно на диаграмму по оси ординат наносятся задачи, а по оси абсцисс дни спринта. Также на диаграмме строится идеальный план работы. После, каждый день наносятся выполненные задачи и сравниваются с идеальным планом работы (рис. 4.5).

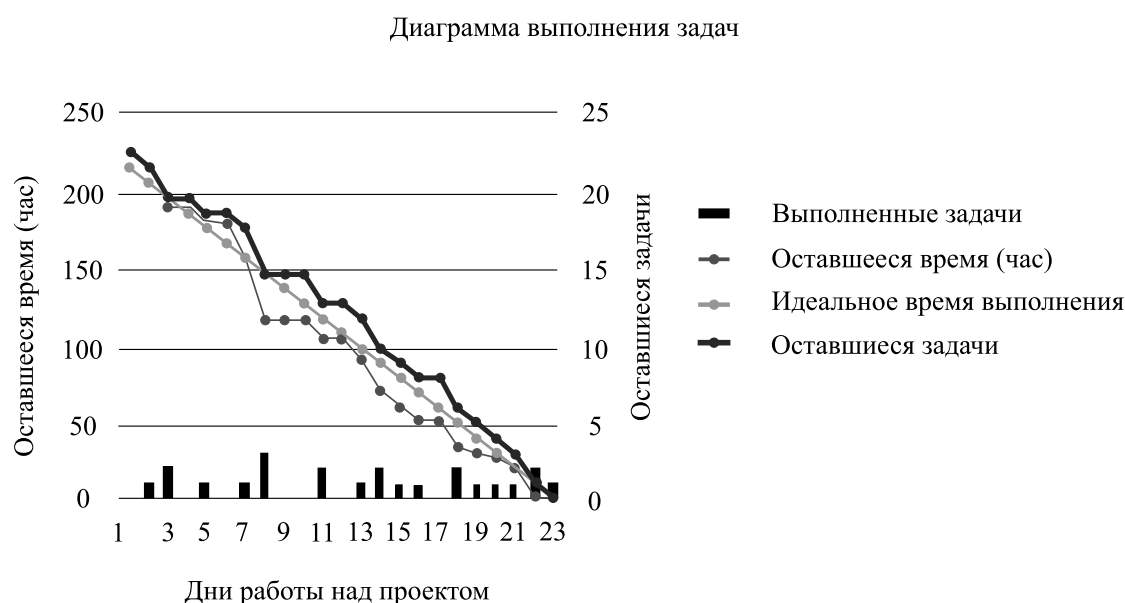


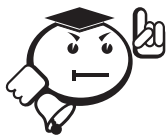
Рис. 4.5 – Пример диаграммы выполнения задач

Для определения трудоёмкости каждой задачи перед спринтом проводится так называемый планировочный покер, целью которого является выяснение интегрального показателя сложности выполнения той или иной задачи. При планировочном покере используется нумерованная колода карт, на каждой карте пишется время, которое необходимо затратить на ту или иную задачу. При доставании задачи из резерва проекта члена проектной команды достают карту с тем количеством часов, за которые он мог бы выполнить задачу. После этого карты раскрываются и участники с наибольшей и наименьшей оценкой высказываются: почему они поставили именно такие оценки. За счёт подобного обсуждения может быть выявлено наиболее оптимальное решение задачи, которое знает более опытный участник команды. Обсуждение сроков продолжается до тех пор, пока не будет достигнуто соглашение. Затем каждой задаче выставляется сложность по выявленной оценке. По этим же оценкам строится идеальный план выполнения задач на диаграмме.

#### Роли в Scrum-методологии.

В процессе, выполняемом по Scrum-методологии, существуют две группы ролей — это «свиньи» и «куры». Эти названия были использованы из-за шутки.





.....

Свинья идёт по дороге. Курица смотрит на нее и говорит: «А давай откроем ресторан!» Свинья смотрит на курицу и отвечает: «Хорошая идея, и как ты хочешь его назвать?» Курица думает и говорит: «Почему бы не назвать «Яичница с беконом»?». «Так не пойдёт, — отвечает свинья — ведь тогда мне придётся полностью посвятить себя проекту, а ты будешь вовлечена только частично».

.....

Основные роли в методологии Scrum «Свиньи»:

- Scrum-мастер — проводит совещания, следит за соблюдением всех принципов Scrum, разрешает противоречия в команде, защищает её от отвлекающих факторов;
- владелец продукта — представляет интересы конечных пользователей и других заинтересованных в продукте сторон;
- Scrum-команда — кросс-функциональная команда разработчиков проекта, состоящая из специалистов разных профилей. Размер команды составляет в идеале  $7 \pm 2$  человека. Никто кроме команды не может быть вовлечён в разработку на протяжении спринта.

Дополнительные роли в методологии Scrum «Куры»:

- пользователи — те, для кого разрабатывается проект;
- клиенты, продавцы — лица, которые инициируют проект и для кого проект будет приносить выгоду. Их вовлекают в Scrum только во время обзорного совещания по спринту;
- управляющие — люди, управляющие персоналом;
- эксперты-консультанты — сторонние эксперты, необходимые для полного понимания предметной области.

### Встречи.

Для решения насущных проблем и управления процессом работы необходимо проводить встречи и совещания. В Scrum-методологии есть несколько определённых типов встреч.

#### Планирование спринта (Sprint Planning Meeting).

Планирование спринта происходит в начале новой итерации спринта.

- Из резерва проекта выбираются задачи, обязательства по выполнению которых берёт на себя команда в процессе спринта.
- На основе задач строится резерв спринта. Задачи оцениваются в идеальных человеко-часах.
- Решение задачи не должно занимать более 12 часов или одного дня. При необходимости задача разбивается на подзадачи.
- Обсуждается и определяется, как и каким образом будут выполнены задачи.
- Планирование спринта ограничивается восьмью часами для того, чтобы не затягивать обсуждение до бесконечности.

**Ежедневное совещание (Daily Scrum Meeting).**

- Начинается точно вовремя, вплоть до того, что могут вводиться символические штрафы на опоздания в виде сладостей или пиццы.
- Длится не более 15 минут и проводится стоя, для того чтобы избежать затягивания совещания.

В течение совещания каждый член команды отвечает на 3 вопроса:

- Что сделано с предыдущего совещания?
- Что будет сделано с текущего совещания до следующего?
- Какие проблемы мешают достижению целей спринта? (Решением этих проблем занимается Scrum-мастер).

**Обзор итогов спринта (Spring Review Meeting).**

Проводится после завершения спринта.

- Команда демонстрирует новую функциональность продукта всем заинтересованным лицам.
- Все члены команды участвуют в демонстрации (один человек может показывать всё или каждый показывает, что сделал за спринт).
- Нельзя демонстрировать незавершённую функциональность.
- Обзор ограничен четырьмя часами в зависимости от выполненных задач и размера продукта.

**Ретроспективное совещание (Retrospective Meeting).**

Проводится после завершения спринта.

- Члены команды высказывают своё мнение о прошедшем спринте.
- Отвечают на два основных вопроса:
  - Что было сделано хорошо в предыдущем спринте?
  - Что надо улучшить в следующем?
- Выполняют улучшение процесса разработки (решают вопросы и фиксируют удачные решения).
- Совещание ограничено одним-тремя часами.

**Достоинства и недостатки.**

Scrum как гибкая методология обладает следующими достоинствами:

- чёткое разделение ролей в команде и принцип невмешательства в работу команды помогает управлять процессом разработки более гибко и точно. Это позволяет не допускать владельцев проекта к продукту в процессе разработки и более гибко контролировать изменения требований;
- короткие спринты работы позволяют быстрее получать обратную связь от владельцев проекта и намного быстрее, чем в водопадной методологии, изменять продукт и исправлять возникающие ошибки;
- кросс-функциональная команда разработки позволяет более устойчиво реагировать на возникновение человеческого фактора (болезни, увольнения и т. д.);

- жёсткие рамки совещаний не позволяют превращать их в бесконечные посиделки «по душам».

Как мы знаем, не существует серебряной пули, решающей все задачи, поэтому Scrum-методология имеет и свои недостатки:

- как ни странно, очень часто говорят о чрезмерной трате времени на встречи, т. к. разработчикам комфортнее работать без отрыва от процесса программирования;
- кросс-функциональная команда не всегда справляется со сложными задачами, т. к. появляется необходимость в человеке с более высоким навыком работы с определённым типом задач, заменить которого на другого члена команды не представляется возможным.

### Экстремальное программирование.

Экстремальное программирование (eXtreme Programming, XP) — это гибкая методология разработки ПО, придуманная Кентом Бекком, Уордом Каннингемом, Мартином Фаулером и другими. Первый проект по этой методологии был запущен 6 марта 1996 года [35–37].

#### Основные приёмы XP.

Все приёмы экстремального программирования можно разбить на четыре основные группы.

*Короткий цикл обратной связи.*



.....  
***Разработка через тестирование (Test Driven Development)*** — разработка ПО, основным движимым действием которой является написание модульных тестов ещё до написания рабочего программного кода [37].  
 .....

До написания кода пишется тест, на который потом пишется рабочий код. Такой подход позволяет сразу писать рабочий программный код, и отпадает необходимость писать модульные тесты после реализации большого количества функций, на что обычно у программиста не остаётся ни желания, ни времени. Подробнее о разработке через тестирование будет написано в главе 9.

*Планировочный покер* — такой же процесс, как и в Scrum.

*Заказчик всегда рядом* — принцип, по которому всегда необходимо иметь в команде представителя заказчика, для того чтобы оперативно решать возникающие вопросы по реализации тех или иных функций. Этот принцип помогает гибко реагировать на изменение требований и быстро получать обратную связь от заказчика.

*Парное программирование* — процесс программирования, при котором над кодом работают одновременно два человека. При этом один разработчик занимается написанием кода, а второй следит за качеством кода и помогает заранее исключить нерабочий или ошибочный код. Обычно в такие пары объединяют программистов разного уровня, чтобы менее опытные разработчики могли перенимать опыт старших коллег. Такой способ программирования позволяет, помимо уменьшения количества ошибок, добиться совместного владения кодом, т. е. за написанный код

может отвечать не только человек, его написавший, но и тот, кто смотрел за написанием.

*Непрерывный, а не пакетный процесс.*



.....  
**Непрерывная интеграция (Continuous Integration)** — процесс разработки ПО, включающий в себя автоматическое выполнение всех шагов по компиляции, тестированию, обфускации кода, сборки установщика или размещения продукта в качестве сервиса на выделенных для этого серверах.  
 .....

Подобный подход позволяет в значительной степени автоматизировать большинство рутинных шагов для более оперативного реагирования на возникающие требования и проверки на ошибки вновь добавленных функций. Непрерывная интеграция подробно описана в главе 9.



.....  
**Рефакторинг** — процесс преобразования рабочего кода для получения более понятных, синтаксически грамотных и менее ошибочных решений.  
 .....

Рефакторинг кода более подробно описан в главе 8.

*Частые небольшие релизы* — подход, согласно которому приоритет отдаётся выпуску продуктов короткими итерациями в 2–4 недели, как в Scrum-методологии.

*Понимание, разделяемое всеми.*

*Простота* — необходимость принятия простых решений является важным свойством программирования. Разработчику и так приходится всё время работать со сложными системами, поэтому поиск простых и лаконичных решений должен являться одним из главных движимых разработчиком постулатов. Необходимость в простых решениях выражена в философии Бритвой Оккама — это принцип, по которому не стоит плодить сущности без необходимости. В программировании придумали следующий шуточный акроним KISS (Keep It Simple Stupid! Сделай это проще, тупица!).

*Метафора системы* — принцип, который основывается на разработке модулей продукта, которые связаны друг с другом в единую систему. Соответственно каждый разработчик должен понимать, что его работа не является отстранённой и должна укладываться в общую систему для выполнения накладываемых системных ограничений.

*Коллективное владение кодом* — принцип, позволяющий гибко перераспределять командные ресурсы на решение различных проектных задач.



.....  
**Стандарт кодирования** — принцип, говорящий о необходимости использования одного общего стандарта кодирования, который отвечает за именование локальных переменных, классов, полей, методов и пр.  
 .....

Такой стандарт кодирования позволит более удобно работать в команде и улучшит коллективное владение кодом, т.к. если поставить разработчика на модуль, написанный без использования стандарта кодирования, он очень много времени потратит на изучение кода и понимание принципов его работы. Помимо этого, заранее прекращаются различные «религиозные войны», в течение которых разработчики могут спорить о правильности именовании локальной переменной с большой или маленькой буквы, но это никак не повлияет на реализацию необходимого заказчику функционала.

*Социальная защищённость программиста.*

*40-часовая рабочая неделя* — принцип, согласно которому программист не должен работать 12 часов 7 дней в неделю. Программист, как человек интеллектуальной профессии, так же устаёт от напряжённого графика и может значительно потерять работоспособность, что скажется как на качестве кода, так и на скорости решаемых задач. Менеджмент команды должен сделать всё возможное для недопущения авральной работы, что может значительно улучшить качество кода и уменьшить количество потенциальных ошибок.

**Достоинства и недостатки.**

Достоинства экстремального программирования:

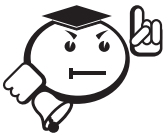
- как и любая гибкая методология, XP позволяет оперативно и гибко реагировать на изменения требований к продукту и быстро получать обратную связь от заказчика. Это приводит к уменьшению затрат на разработку и уменьшению рисков;
- изначально заложенный принцип заботы о разработчике позволяет сказать, что эта методология является для разработчиков более удобной, чем другие.

Недостатки экстремального программирования:

- методология формально не ограничивает сроки итераций, что может привести к неплановому увеличению сроков разработки при неправильном планировании;
- методология не фиксирует необходимые для разработки формальные документы (такие, как резерв проекта или спринта в Scrum), что может привести к появлению различных, зачастую исключаящих документов, регламентирующих разработку;
- использование TDD влечёт за собой и недостатки TDD, т.к. разработку сложных программных систем, с меняющимися программными интерфейсами достаточно сложно вести через тестирование.

**Kanban.**

Одной из современных методологий разработки ПО является методология Kanban, пришедшая к нам из Японии, а именно из производственной системы фирмы Тойота. Методология Kanban берёт своё начало в концепции бережливого производства, где главными управляющими мотивациями при разработке являются методы, сокращающие издержки производства, т.е. уменьшающие затраты на производство и увеличивающие скорость работы команды разработки.



.....

Название методологии пошло от досок Kanban, используемых на предприятиях фирмы Тойота для отслеживания решаемых в данный момент задач. Слово «камбан» по-японски дословно означает «рекламный щит, вывеска», в финансовой среде ошибочно был принят вариант с ошибкой в транскрипции — Kanban. Система Kanban была разработана в Тойоте в 60-х годах. Подробно принципы бережливого производства и система Kanban описаны в книге Тайчи Оно [38].

.....

Основной задачей системы является уменьшение выполняющейся в данный момент работы. Например, вы разрабатываете большую программную систему с множеством функциональных возможностей, возможно, команде захочется взять все задачи и делать их параллельно, что не принесёт никакого решения, а только усложнит процесс, поэтому принимается решение — ограничить количество выполняемых задач на выбранном этапе 10 задачами (такое количество используется и при производстве в Тойоте).

Для того, чтобы явно разграничить процесс производства от разработки ПО, следует выделить следующие особенности методологии:

- необходимо понимать, что Kanban — это не конкретный процесс, а система ценностей, как и Scrum и XP. Соответственно, никто вам не скажет, что и как делать по шагам — вы должны руководствоваться основными правилами методологии для достижения результата;
- обязательно нужно следовать фразе: «Уменьшение выполняющейся в данный момент работы», — т. к. она является основополагающей в методологии;
- Kanban ещё гибче, чем Scrum или XP, поэтому, выбирая данную методологию, следует соотносить её с выполняемым процессом и быть готовым гибко подстраиваться под процесс при необходимости.

Разница между Kanban и Scrum:

- в Kanban нет чёткого ограничения по времени ни на какие этапы;
- в Kanban задачи больше и их меньше;
- в Kanban оценки сроков на задачу опциональные или их вообще нет;
- в Kanban отсутствует такая метрика, как «скорость работы команды», считается только среднее время на полную реализацию задачи.

Достаточно очевидным является вопрос читателя, а как тогда контролировать творческий коллектив разработчиков при отсутствии метрик элементарного контроля: сроков, производительности и пр. В этом случае менеджером команды следует быть честными по отношению к себе, т. к. явного контроля и раньше не было. Большинство используемых метрик являются мнимыми и лишь позволяют получить видимость контроля и примерные планы по ресурсам и срокам для заказчика проекта. Поэтому отсутствие метрик не может значительно ухудшить процесс разработки.

Главным концептуальным отличием Kanban от Scrum является ориентация не на выполнение спринта, а на выполнение задачи. В Kanban задача проходит все

этапы и демонстрируется только после её завершения: именно поэтому в Kanban отдаётся приоритет большим задачам для того, чтобы не засорять пул задач.

Для отслеживания задач в работе используется Канбан доска (рис. 4.6).

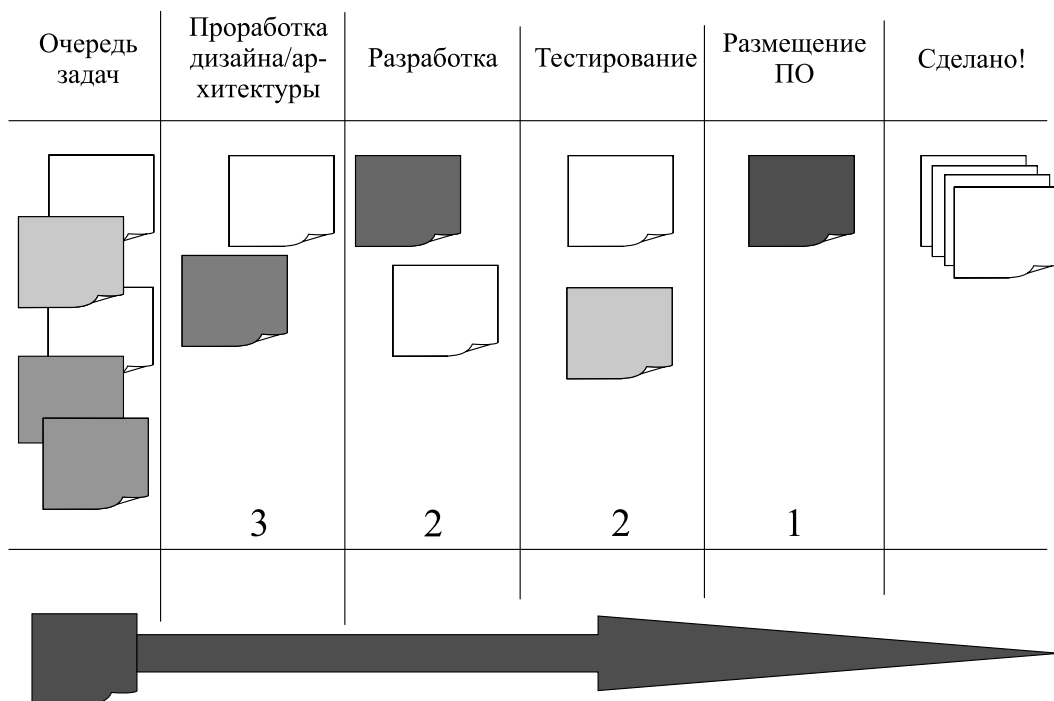


Рис. 4.6 – Пример Kanban доски

Ниже приведено описание столбцов.

*Цели проекта* — необязательный, но полезный столбец. Он ставит перед командой определённые цели, которых нужно будет добиться через решение задач проекта. В качестве таких целей могут выступать следующие: «Увеличить скорость работы алгоритма расчёта электрических цепей на 20%», «Сделать приложение кроссплатформенным для возможности запуска на Windows, Ubuntu и OSx».

*Очередь задач* — тут хранятся задачи, готовые к тому, чтобы начать их выполнять. На верх столбца помещается наиболее приоритетная задача, которая перемещается на следующий этап.

*Проработка дизайна/архитектуры* — этот и остальные столбцы до «Сделано» могут меняться, т.к. у каждой команды есть свои этапы, через которые должна пройти задача до её полного решения.

*Разработка* — тут задача находится до того времени, пока не будет закончена разработка кода для неё. Если на предыдущем шаге были допущены ошибки в архитектуре — задача может быть возвращена на предыдущий этап.

*Тестирование* — в этом столбце задача находится, пока она тестируется. Если найдены ошибки — возвращается в Разработку.

*Размещение ПО (deployment)* — у каждого проекта есть свой процесс размещения, это может быть развёртка сервиса на сервере клиента или комит кода в репозиторий.

*Сделано* — сюда стикер попадает только тогда, когда завершены все работы.

В любой разработке случаются срочные задачи, которые должны быть выполнены быстро. Для таких задач делается отдельный поток через все этапы размерности в одну задачу. Попадая в этот поток, задача становится высокоприоритетной, и команда делает всё, чтобы завершить её как можно скорее.

Теперь перейдём к самому важному — описанию следования концепции бережливого производства. Под каждым столбцом ставится цифра, которая описывает, сколько максимально задач может находиться сейчас в этом столбце. Эта цифра подбирается экспериментально в зависимости от размера команды. Допустим, в команде 6 человек, а в столбце стоит цифра 3 — это значит, что команда разработки будет одновременно работать над 3 задачами и сможет консультироваться друг с другом для решения задачи. Если поставить цифру 1, то большая часть разработки заскучает и либо будет бесконечно обсуждать те или иные задачи, либо начнёт заниматься чем-нибудь другим, но не работой над проектом. Поставив цифру 6, мы получим полностью загруженную команду разработки, однако мы потеряем их способность к обсуждению и нахождению более удачных и оптимальных решений, что может привести к увеличению времени разработки.

Также следует сказать, что задачи на доске — это не просто задачи — это минимальная маркетинговая функциональность, т. е. такая функциональность, которая добавляет ценность вашему продукту и может быть продана клиентам.

Что даёт подобный подход с этапами и лимитами на выполняемые задачи:

- уменьшение числа параллельно выполняемых задач сильно уменьшает время выполнения каждой отдельной задачи. Эта особенность берёт своё начало в природе человеческого мышления, если мы попробуем решать много задач одновременно, а потом возьмём несколько задач последовательно, то это приведёт к значительному сокращению времени работы над задачей;
- сразу видны слабые места команды. Например, если столбец тестирования начнёт переполняться, значит, тестировщики не справляются и необходимо подключить к тестированию разработчиков;
- можно вычислить примерное время выполнения одной задачи, отметив на карточке начало работы над ней. Это позволит хоть как-то посчитать те самые метрики, которые так нужны менеджеру и заказчику. Помимо этого, можно всё время работать над временем выполнения той или иной задачи для того, чтобы увеличить производительность и ускорить процесс разработки.

Весь Kanban можно описать всего тремя основными правилами:

- визуализируйте производство — Kanban доска;
- ограничивайте количество работы в данный момент;
- измеряйте и оптимизируйте время решения задач.

Выше в достаточном количестве перечислены достоинства Kanban, ниже остановимся на видимых недостатках процесса:

- при всей фиктивности имеющихся метрик — заказчик всегда планирует освоить определённое количество ресурсов на разработку и закладывает некоторую единицу времени на выполнение проекта, если этих метрик не будет — вряд ли он согласится на работу с подобной командой;



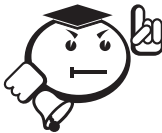
- работа по методологии Kanban требует гибкости и дисциплинированности, при отсутствии этих качеств очень сложно будет работать с проектом;
- отсутствие формальных сроков может восприниматься разработчиками как возможность заниматься любой другой работой, особенно, если задача, стоящая в приоритете, является не интересной, а рутинной. В таком случае хорошо должен сработать менеджмент команды.

### 4.2.3 Другие методологии

#### Cleanroom.

Помимо гибких методологий существуют и другие подходы, которые не настолько быстро подстраиваются под изменения. Однако в некоторых сферах, например разработке высококачественного ПО для нужд космической или военной промышленности, нет большой необходимости в использовании гибких методологий, наоборот, в этих сферах выбирают наиболее формализованные методологии, одним из примеров подобных методологий является Cleanroom.

Название Cleanroom появилось из электронной промышленности, в которой для изготовления современных электронных изделий используются чистые комнаты с ограниченным содержанием пылинок на 1 м<sup>3</sup> воздуха. Также и ПО, разрабатываемое по полностью формализованным подходам, имеет небольшое количество ошибок на определённое количество строк кода.



.....  
 В нескольких проектах — например, в проекте разработки ПО для космических кораблей — благодаря сочетанию методик формальной разработки, обзоров кода коллегами и статистического тестирования был достигнут такой уровень, как 0 дефектов на 500 000 строк кода [12].  
 .....

Методология была разработана Харланом Миллзом и несколькими его коллегами из IBM.

#### Используемые принципы.

*Разработка ПО с использованием формальных методов* — большинство процессов и подходов, используемых в проекте, подвергаются строгому формальному контролю. Для каждого модуля полностью описываются спецификации, его входы и выходы, используемые структуры данных, используемые алгоритмы и их особенности. Все части программы проходят формальную проверку на соответствие спецификациям. Подобный метод формальной проверки всех спецификаций называется Методом Структурного Ящика.

*Инкрементальное добавление функциональности под управлением статистического контроля качества* — разработка с помощью Cleanroom использует итеративный подход к разработке, при котором каждая новая функция поочерёдно добавляется в продукт. При прохождении каждой итерации проводится статистическое тестирование, с помощью которого по формальным методикам измеряется качество ПО. Полученные метрики сравниваются с заданными для данного этапа,

если полученные результаты не удовлетворяют заданным для этого этапа стандартам, то проект возвращается на стадию проектирования.

*Статистическое тестирование* — тестирование продукта проводится на статистических данных, это подразумевает генерацию всех вариантов наборов входных тестовых данных для того, чтобы покрыть все возможные случаи некорректной работы программного модуля. Для генерации подобного набора данных используют Statistical Test Generation Facility (STGF), который имеет собственный язык описания тестовых данных, позволяющий запрограммировать сценарий тестирования — характер распределения данных, моменты возникновения критических событий и т. д. В результате STGF генерирует код на языке C, который после компиляции и запуска подает на вход тестируемой программы тестовые данные.

*Командная разработка* — Cleanroom предполагает, что разработка ведётся небольшими группами (от 6 до 8 человек). При этом предусматривается парный контроль, но не исключается индивидуальная разработка.

Перераспределение времени между этапами жизненного цикла — поскольку одной из главных целей Cleanroom является предотвращение ошибок, времени на этап проектирования отводится существенно больше по сравнению с другими технологиями. Cleanroom не предполагает увеличения общего времени разработки проекта, но делает акцент на проектировании и верификации. Схема разработки по методологии «чистой комнаты» приведена ниже (рис. 4.7).



Рис. 4.7 – Схема разработки по методологии «чистой комнаты»

### Преимущества и недостатки.

В качестве преимуществ можно выделить следующие пункты:

- корректность, надёжность и понятность программного продукта, достигающаяся использованием формальных методик контроля разработки ПО;
- данная технология минимизирует затраты компании-разработчика на стадии поддержки программного продукта за счёт увеличения затрат на стадии разработки. Как правило, такой подход в целом снижает совокупные затраты разработчика на всем сроке жизненного цикла продукта.

В качестве недостатков можно выделить следующие:

- работа по формальным методикам может не подходить большинству коммерческих продуктов и некоторым командам разработки из-за особенностей рынка (необходимости быстрого наращивания функционала в условиях изменяющихся требований) или особенностей работы конкретной команды (минимальные затраты на этапы проектирования и верификации ПО);
- большое количество документации может потребовать значительное количество технических писателей в команде, что может существенно замедлить разработку из-за большого количества коммуникаций и необходимости выполнения большого количества рутинной работы по описанию спецификаций к ПО.



## Выводы

В данной главе представлены не все имеющиеся методологии разработки ПО, приведён краткий обзор лишь некоторых из них. Если читатель захочет подробно познакомиться с каждой из методологий — ему следует обратиться к специализированной литературе по тематике.

В качестве выводов хотелось бы сказать, что не существует «серебряной пули» для каждого проекта, в любом случае приходится искать компромиссы, и поступаясь одними метриками разработки ПО, например количеством ошибок, качественно усиливать другие, например скорость выпуска релизов.

Также не следует догматично следовать каждой методологии и канонически использовать все её свойства и инструменты: каждый проект уникален по-своему, поэтому под каждый проект и каждую команду следует подстраивать процесс разработки ПО, внедряя новые принципы и совершенствуя существующие.



## Контрольные вопросы по главе 4

1. Что такое методология разработки ПО и зачем она нужна?
2. Для каких видов проектов удобно использовать водопадную методологию?
3. В чём выражается «гибкость» гибких методологий разработки?
4. Какая из методологий является наиболее выгодной для разработчика из-за своей социальной программы?
5. Какая из методологий позволяет добиться наименьшего количества ошибок в расчёте на количество строк кода и за счёт чего это достигается?

---

## Глава 5

# ПОЛЬЗОВАТЕЛЬСКИЕ ИНТЕРФЕЙСЫ

---

Пользовательский интерфейс играет решающую роль в вопросе, будет ли конечный пользователь использовать ваш программный продукт или нет. Для пользователя всегда сокрыты внутренняя архитектура программы, качество кода или покрытие этого кода тестами. Ему не важно, используете ли вы объектно-ориентированную модель или процедурную. Пользователь работает с программой через интерфейс, и если этот интерфейс неудобен, пользователь откажется от вашего продукта.

Таким образом, особое внимание при разработке необходимо уделить взаимодействию пользователя и продукта. Это специальная и весьма специфичная область проектирования, так как связана с некоторым неформализуемым понятием комфорта пользователя. Здесь нет каких-либо строгих правил и алгоритмов, позволяющих создать эффективное решение. Можно лишь определить набор рекомендаций, следование которым определяется личным опытом разработчика. Если вы планируете разработку серьезного коммерческого проекта, то для этой цели вам лучше нанять опытного проектировщика интерфейсов. В данной же главе мы рассмотрим лишь общие понятия и принципы проектирования пользовательских интерфейсов, которые должен знать любой разработчик.

В первую очередь нужно понять, какую именно цель преследует пользовательский интерфейс. При написании технического задания или проектировании архитектуры мы представляем программу, как черный ящик, — некоторая коробка, о внутреннем представлении которой мы ничего не знаем. Всё, что мы знаем, это какие входы и выходы у этого ящика и какую задачу этот ящик решает. Современные программные решения достаточно большие и, как правило, решают не одну задачу, а сразу множество задач, предоставляя целый набор функциональности для пользователя. Но вся эта функциональность сокрыта внутри программы, и получить к ней доступ можно только через входы/выходы. Интерфейс должен предоставить пользователю возможность комфортно для себя использовать данную функциональность. Таким образом, цель пользовательского интерфейса — организация

ввода/вывода приложения для достижения максимальной эффективности работы пользователя.

Эффективность работы можно оценить через минимальное количество действий, необходимых пользователю, для решения своей конкретной задачи. То есть, чем меньше действий сделает пользователь, чтобы решить свою проблему, тем эффективнее его работа в программе. Если же пользователю приходится выполнять огромное количество рутинной работы, связанное, например, с многочисленными переходами между окнами или вводом данных в поля, не относящихся непосредственно к его задаче, то эффективность его работы значительно снижается.



.....

Следовательно, при проектировании интерфейса необходимо решить следующую задачу — минимизация действий пользователя в решении своих задач. Примером может послужить ситуация, когда изначально пользователю требовалось в строго определенном порядке пройти по трём окнам приложения, вводя различные данные, а ваше новое решение позволило организовать ввод всех данных в одном простом окне.

.....

Однако здесь есть некоторый подводный камень. Как мы уже говорили, современные приложения решают не одну задачу, а множество задач, и минимизация действий пользователя для решения одной задачи может приводить к увеличению действий пользователя для решения другой задачи. В итоге, чтобы прийти к некоторому компромиссу, разработчики создают интерфейс, по сложности напоминающий кабину космического аппарата, разобраться в которой рядовому пользователю будет затруднительно.



.....

Правильным решением данной проблемы является расстановка приоритетов среди тех задач, которые пользователь будет решать с помощью вашей программы. Вам необходимо определить те задачи, которые составляют 80% всего объёма работ пользователя, и сконцентрироваться на проработке интерфейса для решения *этих* задач. Оставшиеся 20% не так важны для пользователя, следовательно, интерфейс должен учитывать данные варианты использования в меньшей степени.

.....

## 5.1 Правила верстки пользовательского интерфейса

Пользовательский интерфейс состоит из отдельных элементов и форм, которые собираются в единое целое. Проектирование интерфейса заставляет думать не только о расположении элементов, но и о динамике перехода пользователя от одного подобного элемента к другому таким образом, чтобы это было максимально удобно и эффективно. Это нетривиальная задача, и для её решения необходимо понимать, как именно пользователь будет действовать при работе с программой.

Однако прежде чем обсуждать поведение пользователя, необходимо поговорить о грамотном представлении отдельных элементов и форм. В частности, о взаимном расположении элементов пользовательского интерфейса — верстке.

Основные правила верстки:

1. Определение приоритетов функциональности.
2. Определение размеров элементов.
3. Группировка.
4. Выравнивание и отступы.
5. Отрицательное пространство.

*Определение приоритетов функциональности.* В первую очередь необходимо определить, какие элементы играют более важную роль для пользователя на данной форме, а также в каком порядке пользователь будет с ними взаимодействовать. Очень часто бывает, что пользователю нужны не все элементы формы, а только их часть, остальные же играют вспомогательную роль и требуются в нечастых сценариях использования. Следовательно, такие второстепенные элементы управления должны быть менее заметны на форме, либо даже скрыты в дополнительной раскрываемой панели, например Дополнительные настройки (Advanced). Это позволит пользователю быстрее разобраться в управлении программой и сконцентрироваться на решении более частых задач. Также важно выяснить, в каком порядке пользователю будет комфортнее заполнять те или иные элементы управления и расположить их на форме в соответствующем порядке. Как правило, человеку удобнее читать и заполнять элементы слева направо сверху вниз, но бывают и исключения.

Например, в формах регистрации всегда есть обязательные и необязательные поля. При первом знакомстве с программой или сайтом многие пользователи, как правило, пропускают необязательные поля. Следовательно, их можно опустить ниже обязательных полей, сделать их менее заметными или скрыть их в панели «Дополнительная информация» (рис. 5.1).

Также, заполняя информацию о себе, пользователю удобнее начать с фамилии, имени и отчества и лишь затем перейти к дате рождения или своей средней школе. Поэтому фамилия и имя зачастую выносятся на верх регистрационной формы.

*Определение размеров элементов.* Каждый элемент предназначен либо для ввода, либо для вывода данных. В любом случае, эти данные имеют определенный объём, и размеры элемента должны быть определены исходя из данного объёма (рис. 5.2).

Например, возраст человека вряд ли может превысить трехзначное число. Следовательно, поле для ввода возраста не должно быть больше, чем надо для отображения трехзначных чисел. Точно также поле ввода зарплаты может быть рассчитано на шестизначную сумму и номер телефона всегда состоит из 11 цифр. Учтите это в своём интерфейсе.

*Группировка.* Элементы с общим назначением должны быть сгруппированы вместе. Это облегчит навигацию для пользователя, сделает использование формы более комфортной.

Для формы регистрации можно разделить всю информацию о человеке на несколько групп:

- Основная — фамилия, имя, отчество, пол, дата рождения.
- Образование и работа — средняя школа, университет, специальность, текущее место работы.
- Дополнительная — увлечения, биография, фотографии и т. д.

Registration

Surname:

Name:

\*E-mail:

Sex:  Male  Female

Birthdate:

\*Password:

Description:

Confirm

a)

Registration

\*E-mail:

\*Password:

Advanced Info (optional)

Surname:

Name:

Sex:  Male  Female

Birthdate:

Description:

Confirm

б)

Рис. 5.1 – Формы регистрации: а) обязательные поля перенесены вверх, б) необязательные поля вынесены в отдельную панель

Surname:

Age:

Description:

a)

Surname:

Age:

Description:

б)

Рис. 5.2 – Формы регистрации: а) поля фамилии и описания формы слишком маленькие для комфортного заполнения, а поле возраста слишком большое; б) недостатки устранены

Отдельным преимуществом данной группировки является то, что теперь мы выделили одну группу элементов «Основная», с полями, обязательными для заполнения, переместив необязательные поля во второстепенные группы.

*Выравнивание и отступы.* Ничто так не портит впечатление от программы, как небрежно разбросанные элементы управления. Пользователь сразу перестает до-

верить программному продукту, он будет ощущать дискомфорт при работе с такой программой (см. рис. 5.3).

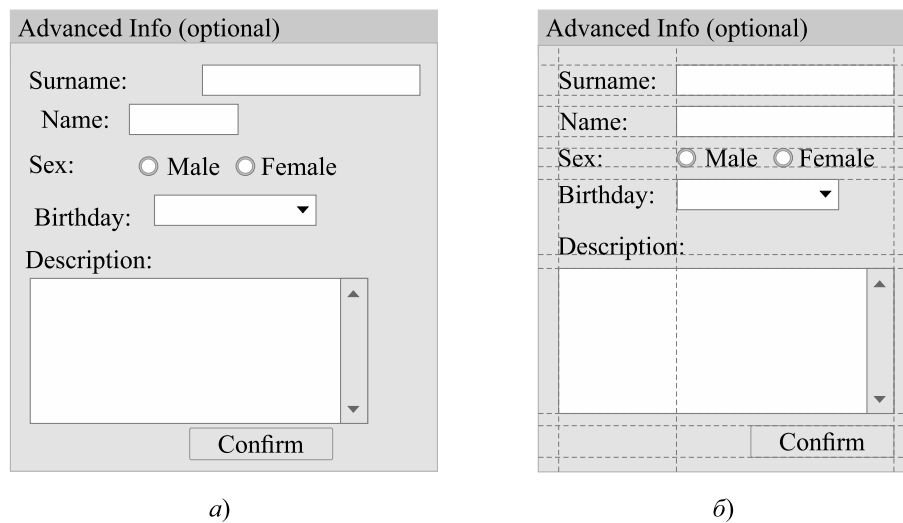


Рис. 5.3 – Формы регистрации: а) элементы слева набросаны на форму хаотично; б) элементы слева выровнены по мысленным направляющим с соблюдением одинаковых отступов

Мысленно наметьте на форме направляющие линии, по которым вы будете выравнивать пользовательские элементы. При этом старайтесь выравнивать элементы по левому или правому краю, выравнивание элементов по центру чаще всего не воспринимается пользователем. Определитесь с величиной отступов у полей или между элементами. Например, вы можете решить, что все поля должны составлять 10 пикселей, отступы между группами элементов 5 пикселей, а между элементами внутри группы 3 пикселя. В дальнейшем, старайтесь придерживаться данных правил для *всех* форм приложения. Это создаст некое единообразие оформления.

*Отрицательное пространство.* Понятие отрицательного пространства, так же, как и многие другие понятия верстки, пришло из типографии. Под данным понятием подразумевается, что не стоит пытаться как можно более компактно расположить элементы управления на форме. Разумеется, полезное пространство должно быть использовано максимально эффективно. Однако если пользовательские элементы будут расположены слишком плотно, пользователю будет слишком тяжело воспринять форму. Между элементами должно быть так называемое отрицательное пространство — свободная область, визуальнo разделяющая элементы. Отступы между элементами являются примером отрицательного пространства, однако иногда необходимо оставлять пустыми целые области формы, чтобы пользователю было комфортнее работать (см. рис. 5.4).

Помимо данных правил следует назвать еще одно — проверка ввода данных пользователем. Оно не относится непосредственно к верстке, однако оно более важно для соблюдения, чем многие другие правила.

Как уже было сказано, пользователь использует формы для ввода или вывода данных. При вводе данных пользователю легко ошибиться, например, нажав не ту клавишу. В итоге, имя пользователя может содержать цифру или точку, а зарплата может содержать букву. Если пользователь вовремя не заметил ошибку, неправиль-



ные данные попадут в программу, что может привести к её неправильной работе. Ваша задача как разработчика — не допустить неправильного ввода данных пользователем, и лучшим решением данной задачи будет проверка данных уже на этапе ввода.

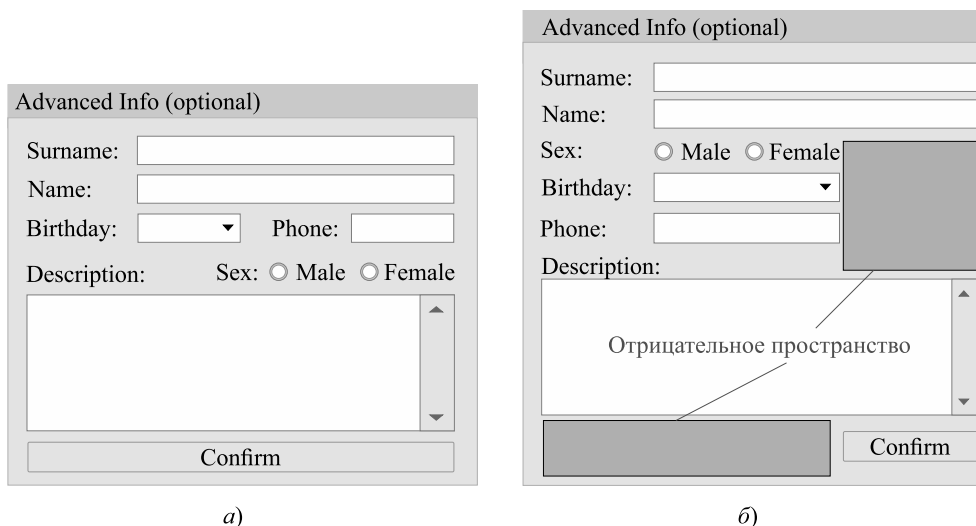


Рис. 5.4 – Форма справа б) выглядит более последовательной и удобной для заполнения по сравнению с нагромождением элементов слева а) благодаря отрицательному пространству



## Пример

Например, пользователь вводит в программу свой возраст с клавиатуры. Так как возраст — это всегда число, можно сделать проверку ввода и отсеивать нажатия всех других клавиш, кроме цифр. Таким образом, при нажатии буквы или знака минуса данный символ просто не будет вводиться в поле.

Другой пример, когда пользователь ввел в качестве числа 0. Мы не можем запретить ввод нуля с клавиатуры, так как возраст человека может содержать 0, например, 10, 20 и т. д. Однако и нулевой возраст у человека также быть не может. Или если пользователь ввел 478 в качестве своего возраста — вероятно, он где-то привирает. В таких случаях можно реализовать дополнительную проверку, в которой после нажатия клавиш внутри поля программа будет анализировать введённое число и выводить соответствующее сообщение. Главный принцип этого правила — не дать пользователю ввести неправильные данные либо сообщить о неправильном вводе как можно раньше.

## 5.2 Шаблоны пользовательского поведения

Теперь, когда мы знаем правила оформления отдельных окон и пользовательских элементов управления, мы можем перейти к проектированию интерфейса всего приложения.

Как уже отмечалось ранее, достаточно сложно узнать, какой именно интерфейс будет удобен для пользователя, а какой будет вызывать дискомфорт. Каждый человек индивидуален, и создать что-то, что понравилось бы любому, практически невозможно.



.....  
*Однако в поведении всех пользователей можно выделить определенные закономерности — **шаблоны поведения**.*  
 .....

Если спроектированный интерфейс будет удовлетворять данным шаблонам, пользователь легко освоит вашу программу.

В поведении пользователя можно выделить 12 шаблонов [40]:

1. Безопасное исследование (Safe Exploration).
2. Мгновенное вознаграждение (Instant Gratification).
3. Разумная достаточность (Satisficing).
4. Изменения на полпути (Changes in Midstream).
5. Отложенный выбор (Deferred Choices).
6. Пошаговое построение (Incremental Construction).
7. Привыкание (Habituation).
8. Пространственная память (Spatial Memory).
9. Проспективная память (Prospective Memory).
10. Организованное повторение (Streamlined Repetition).
11. Только клавиатура (Keyboard Only).
12. Советы других людей (Other People's Advice).

*Безопасное исследование.* Хорошее программное обеспечение позволяет людям пробовать неизвестные функции и возвращать систему в исходное состояние, снова пробовать что-то новое и так далее. Если пользователь не уверен, что сможет вернуть изменения обратно, он всегда будет использовать новую функциональность с некоторой опаской. Потеря же введенных данных или неочевидно всплывающие или закрывающиеся окна программ — всё это приводит к отпугиванию пользователя. Даже простой досады достаточно, чтобы пользователь отложил изучение вашей программы на более позднее время [40].

Например, вы разрабатываете редактор текста. Пользователь хочет попробовать поменять стиль текста, изменить цвет, размер, шрифт, переставить отдельные элементы местами. Если результат ему не нравится, он нажимает кнопку «Отмена», и текст возвращается в исходное состояние. Пользователь вновь может экспериментировать с функциями программы, не беспокоясь о потере данных.

*Мгновенное вознаграждение.* Данный шаблон заключается в том, что человек любит видеть результаты своих действий сразу. И если этот результат положительный, пользователь будет доволен. Следовательно, интерфейс программы нужно проектировать таким образом, чтобы с первых же минут работы с приложением пользователь получил простой, но приятный опыт. Для этого попытайтесь предугадать первые действия пользователя в вашей программе и сделать их максимально простыми. Когда пользователь выполнит их и поймет насколько это просто, он почувствует уверенность и будет готов к дальнейшему изучению программы.

*Разумная достаточность.* Данный шаблон подразумевает, что пользователь скорее выберет *достаточно хорошее* или *удовлетворяющее* его решение, а не *наилучшее*, если изучение всех альтернативных вариантов может потребовать траты времени и сил. В рамках программных интерфейсов это означает, что пользователь не будет внимательно изучать каждую кнопку вашего приложения, прежде чем сделать вывод: «Да, эта кнопка делает именно то, что мне нужно». Пользователь просто выберет первую попавшуюся кнопку, которая, на его взгляд, решает данную проблему. Если же эта кнопка впоследствии не сделает ожидаемого действия, пользователь будет разочарован.

Вследствие данного шаблона старайтесь не перегружать формы огромным количеством функций, а сами функции обозначайте максимально доступным образом — короткой понятной текстовой меткой или однозначно интерпретируемой пиктограммой.

*Изменения на полпути.* Очень часто бывает, что человек, выполняющий какой-нибудь процесс, прерывается на середине и переключается на другую задачу. Например, он мог найти новую функциональность на одном из промежуточных шагов и решил её исследовать, отложив исходную проблему. В любом случае, первоначальные цели пользователя могут измениться, а значит, ему нужно предоставить такую возможность — откатиться на несколько шагов назад, перейти к главному окну или какому-либо другому, но при этом, по возможности, не потерять данные, если пользователь захочет вернуться к исходной цели.

*Отложенный выбор.* Данный шаблон подразумевает, что пользователь с большой долей вероятности пропустит незначительные на его взгляд шаги для достижения более важной цели. Например, формы регистрации на сайтах, в частности интернет-магазинах, зачастую бывают излишне подробными. Как правило, пользователь хочет побыстрее пройти регистрацию и перейти непосредственно к покупке. Зачем ему заполнять все эти поля? Возможно, лучше потребовать от него заполнения только нескольких обязательных полей, а затем, когда у пользователя будет время, предоставить ему возможность заполнить оставшиеся данные.

Рекомендациями к данному шаблону могут быть следующие действия:

- Не задавайте пользователю большого числа вопросов с самого начала.
- Минимизируйте количество обязательных полей и визуально их помечайте.
- Скрывайте наименее важные настройки или функции в раскрывающиеся панели или дополнительные окна, если данные настройки являются редко используемыми.
- Используйте заданные по умолчанию значения полей там, где это возможно.

- Предоставьте пользователю возможность вернуться к отложенным полям позже. Возможность отложить заполнение тех или иных полей должна быть очевидна для пользователя.

*Пошаговое построение.* Как правило, достаточно сложно выполнить какую-то мало-мальски серьезную работу за один раз. Человек решает любую задачу постепенно. После выполнения одного шага пользователю необходимо оценить, правильно ли он его сделал, возможно, внести поправки и только потом перейти к следующему шагу. Иногда пользователю даже приходится начинать всё сначала. Постарайтесь разделить выполнение сложных задач на элементарные действия, постоянно демонстрируя пользователю результат работы.

*Привыкание.* Когда пользователь долго работает в одном интерфейсе, многие действия, особенно простые и элементарные, становятся рефлексивными, и человеку уже не нужно задумываться об их выполнении — он делает их машинально.

Что важно, если пользователь начинает использовать другое приложение, он автоматически будет пробовать уже привычные действия в новом интерфейсе. И если новый интерфейс отвечает пользователю ожидаемыми им действиями, взаимодействие с программой значительно упрощается. Обратный же эффект, если пользователь не видит реакции программы или, что еще хуже, в новом интерфейсе привычная и безопасная последовательность действий приводит к потере данных.

Примером могут послужить горячие клавиши. Для многих пользователей стало привычным, что комбинация Ctrl+S сохраняет документ, Ctrl+C копирует, а Ctrl+V вставляет скопированные данные. Если вы в своей программе назначите на данные комбинации другие действия, кардинально отличающиеся от общепринятых, это создаст негативный опыт у пользователей.

Другой пример, пользователь всегда ожидает увидеть главное меню программы в верхней части экрана. Если вы перенесете меню вниз или вправо, это затруднит использование программы для пользователя.

Выполнение данного шаблона очень важно, всегда старайтесь найти те решения пользовательского интерфейса, которые наиболее привычны для большинства людей. Также данный шаблон означает, что всегда нужно быть осторожным к нововведениям в интерфейсе — они могут быть восприняты в штыки.

*Пространственная память.* Зачастую, при работе с множеством объектов или документов, люди ориентируются на воспоминания об их расположении. Как пример, пользователь запоминает расположение иконок на рабочем столе Windows, что в дальнейшем позволяет ему не вчитываться в ярлыки, а просто запускать программы, основываясь на их взаимном расположении. Более того, пользователи сами группируют и перемещают иконки рабочего стола так, чтобы быстрее и удобнее запомнить их расположение.

Постарайтесь следовать следующим рекомендациям:

- Располагайте распространенные элементы (такие как кнопки Ok и Cancel, главное меню, строка состояния и т. д.) в предсказуемых местах.
- Если интерфейс состоит из множества панелей, позвольте пользователю настроить их на своё усмотрение. Пусть программа запомнит это расположение.

- Избегайте скрывания панелей и других элементов. Для пользователя может стать неприятным сюрпризом, когда панель, которую он видел несколько секунд назад, вдруг исчезла.

*Перспективная память.* Психологический феномен, используемый людьми для напоминания себе о запланированных действиях. Например, если вам нужно отправить письмо через несколько часов (а не прямо сейчас), вы можете оставить себе записку и наклеить её на монитор. Другой пример, когда вам в дальнейшем понадобится вернуться к написанию документа на компьютере, вы можете оставить окно редактора открытым, написав большими заметными буквами напоминание внутри самого документа. Суть явления в том, что вы оставляете себе некоторое напоминание, на которое вы среагируете в нужный момент.

Таким образом, если пользователь приступает к задачам и оставляет их незавершенными, подумайте о том, как оставить на видном месте какое-нибудь напоминание о незаконченных задачах.

*Организованное повторение.* В сложных программных комплексах часто бывает, что одну и ту же операцию пользователю приходится повторять многократно. Это может быть обработка фотографий одним и тем же фильтром, или переименование файлов по определенному шаблону. Подобные рутинные действия утомляют пользователя.

Постарайтесь придумать некоторый механизм, позволяющий повторять некоторую последовательность действий многократно. Пример, известная функция «Найти и Заменить» — при однократном нажатии кнопки «Найти» и «Заменить» программа заменит лишь одно найденное слово. Замена же всех слов в документе может стать долгим процессом. Простая кнопка «Заменить все» значительно упрощает эту рутину.

Более сложным механизмом является возможность записи пользователем собственных макросов — небольших скриптовых подпрограмм. Пользователю достаточно один раз выполнить некоторую процедуру и записать её в макрос, чтобы в дальнейшем легко её вызывать одним нажатием клавиши. Подобная функциональность очень удобна в сложных программных комплексах.

*Только клавиатура.* Многие пользователи стараются сделать свою работу за компьютером более эффективной. Одним из средств повышения эффективности является запоминание горячих клавиш, что может сократить время работы с программой в разы. Постарайтесь продумать способ взаимодействия с программой с помощью клавиатуры. Помните, что комбинации клавиш должны легко запоминаться пользователем.

*Советы других людей.* Советы других людей могут сыграть ключевую роль в принятии решений человеком. Даже при изучении новой программы пользователь с радостью будет слушать другого человека, нежели обратится к руководству пользователя. Данный шаблон не имеет каких-то конкретных рекомендаций, и в каждом приложении он учитывается по-разному. Это может быть интеграция в программу социальных сетей, или организация некоторого сообщества пользователей, или глобальная таблица рекордов, если вы разрабатываете компьютерную игру. Однако возможность взаимодействия с другими людьми — важный аспект, который стоит держать в голове при проектировании пользовательских интерфейсов.

Не обязательно учитывать все вышеперечисленные шаблоны поведения в своём интерфейсе. Зачастую, это даже невозможно, так как многие из них противоречат друг другу. В каждом случае вам необходимо прочувствовать самого пользователя, стать на его место и сделать программу так, как это было бы максимально удобно для него.

## 5.3 Прототипирование

Мы обсудили с вами разработку пользовательских интерфейсов с точки зрения важности для пользователей. Однако данный этап играет огромную роль в процессе проектирования. Проектирование интерфейса должно выполняться либо между, либо параллельно с этапами написания технического задания и проектирования архитектуры. Техническое задание станет гораздо понятнее для системных архитекторов и программистов, когда в нём появятся изображения конечного интерфейса. Хороший макет интерфейса может заменить несколько страниц текстового описания и помогает более грамотно и четко сформировать требования к продукту. Данный этап называется *прототипированием*.



.....  
**Прототипирование** — процесс создания **прототипа** программы — макета программы с целью проверки пригодности предлагаемых концепций, как интерфейсных, так и архитектурных.  
 .....

Под прототипом можно понимать очень многое — от набросков окон приложения до небольшой рабочей версии программы, выполняющих простейшую базовую функциональность. Во многих случаях, для демонстрации примерной работы приложения в прототипах используют *псевдореальные* данные — сгенерированные случайным образом, но близкие к реальным.

Разработка прототипа состоит из следующих шагов:

1. Разработка базового варианта прототипа с учетом начальных требований, указанных в техническом задании.
2. Демонстрация прототипа заказчику и фокус-группе целевых пользователей, получение обратной связи с замечаниями и дополнениями.
3. Исправление прототипа, согласно замечаниям, и повторная демонстрация прототипа целевой группе. Данный шаг повторяется до полного согласования интерфейса и выяснения всех необходимых аспектов.

С помощью прототипирования можно решить следующие задачи:

- Прояснение и завершение процесса формулировки требований — обратная связь с пользователями позволяет точно понять, какой продукт они хотят получить и как он должен работать.
- Исследование альтернативных решений — после демонстрации прототипа пользователям концепция приложения может измениться кардинально. Поэтому этап прототипирования также хорошо подходит для проведения так называемого А/В тестирования — когда пользователям предлагают несколь-

ко вариантов интерфейса или его частей и по результатам обратной связи выбирают одну из альтернатив.

- Создание конечного продукта — прототип как черновая версия программы в дальнейшем может дорабатываться и обрывать функциональностью до тех пор, пока не будет получен конечный продукт.

Основной же целью прототипирования является *устранение неясностей на ранних стадиях разработки* [41].

Прототипы можно классифицировать:

- на горизонтальные и вертикальные;
- одноразовые и эволюционные;
- бумажные и электронные.



.....  
**Горизонтальный (поведенческий) прототип** — демонстрирует лишь верхний слой программы — пользовательский интерфейс — без реализации остальных уровней архитектуры.  
 .....

Позволяет пользователям исследовать поведение системы и выяснить, смогут ли они решать свои задачи с помощью данной системы.



.....  
**Вертикальный (структурный) прототип** — выполняется для проверки архитектурных концепций, затрагивает все уровни реализации.  
 .....

Фактически, вертикальные прототипы используются не столько для оценки качества интерфейса, а сколько для оценки эффективности алгоритмов и идей, заложенных в системе. Позволяют оценить время выполнения и производительность критичных мест системы.



.....  
**Одноразовый (исследовательский) прототип** — прототип, предназначенный для быстрого получения ответов на вопросы, необходимых для четкого определения требований к ПО.  
 .....

Прежде чем создавать прототип, необходимо четко представлять, для чего вы хотите его сделать. Если вы решили, что после выполнения задачи прекратите работу с прототипом, т. е. черновым макетом, разрабатывайте быстрый и максимально дешевый макет. Пренебрегайте устойчивостью, надежностью и производительностью. Одноразовый прототип нужен для получения ответа на четко поставленный вопрос, после его разрешения прототип не требует какого-либо сопровождения. Более того, убедитесь, что ни одна часть такого прототипа не попадет в конечный продукт — такой низкокачественный код может стать причиной многих проблем.

Противоположный подход — *эволюционный прототип*. Суть разработки такого прототипа заключается в создании базовой версии программы как основы для разработки конечного продукта. Таким образом, для эволюционного прототипа необходимо сразу же продумывать взаимодействие компонентов и следить за качеством кода. Такой подход требует большего времени на реализацию, что увеличивает риски, связанные с изменением концепции приложения в результате первичных отзывов фокус-группы. Незначительные изменения учитываются в следующей итерации, но если изменения слишком серьезные — прототип придется выбросить с осознанием потраченного времени.

*Бумажный прототип* — вероятно, самый быстрый и дешевый способ прототипирования, когда проектировщик выполняет наброски окон или элементов управления на бумаге, не заботясь о том, где точно будут располагаться те или иные элементы. Элементы по мере необходимости сопровождаются заметками об их поведении. Данная работа может выполняться в присутствии заказчика или фокус-группы, которые могут тут же высказывать собственное мнение и вносить поправки. В отличие от *электронных прототипов*, создаваемых в специальных компьютерных средах прототипирования или даже средах разработки ПО, бумажные прототипы легко поддаются изменению, хотя и не позволяют «прочувствовать» пользователям всю динамику взаимодействия с программой.

Для создания прототипов существует огромное количество готовых инструментов, для разных платформ и устройств. Нет большого смысла перечислять их здесь, так как для прототипирования может использовать даже обычный графический редактор или программы из пакета Microsoft Office. Есть и более специализированные, предоставляющие готовые шаблоны пользовательских элементов. Ну и разумеется, такие среды разработки, как Visual Studio, также позволяют создать черновые макеты, максимально схожие визуально с конечным продуктом и реализующие какую-либо функциональность.

*Оценка прототипа*. Как уже неоднократно говорилось выше, прототип создается с целью ответа на определенные вопросы, возникшие на этапе формирования технического задания. Поэтому, прежде чем демонстрировать прототип фокус-группе, обязательно проработайте список вопросов, на которые им необходимо ответить. В случае горизонтальных прототипов создайте сценарии — некоторую последовательность действий, которую пользователю необходимо будет пройти, и по мере прохождения выясняйте необходимую информацию. Список возможных вопросов:

- Реализует ли прототип все необходимые функции так, как вы этого ожидали?
- Не упущены ли какие-либо необходимые функции?
- Заметили ли вы какие-либо условия ошибки, не учтенные в прототипе?
- Нет ли в прототипе каких-либо ненужных функций?
- Насколько логичной и полной вам кажется навигация?
- Не оказалось выполнение каких-либо задач слишком сложным?

*Руководства*. В данной главе были представлены достаточно обобщенные подходы и техники прототипирования и проектирования пользовательских интерфейсов. Для более подробного изучения данной темы предлагаем ознакомиться с представленным списком используемой литературы. Однако для различных платформ



и для различных устройств есть свои особенности проектирования интерфейсов. Это очевидно, так как интерфейс, придуманный для персонального компьютера, будет мало удобен для планшетов, смартфонов или игровых приставок. Для ознакомления с этими нюансами многие разработчики платформ предлагают ознакомиться с так называемыми руководствами (guidelines). В них перечислены основные концепции и правила интерфейсов целевой платформы, что позволяет достичь единообразия среди всех программ данной платформы. Примером таких руководств могут послужить руководства по интерфейсам iOS или Android.



## Контрольные вопросы по главе 5

1. Что такое прототип?
2. Какие виды прототипов бывают и для чего они предназначаются?
3. Для чего необходимо учитывать шаблоны пользовательского поведения?
4. Что такое безопасное исследование?
5. Что такое разумная достаточность?

---

## Глава 6

# ДОКУМЕНТАЦИЯ

---

Как уже было рассказано в предыдущих главах — корректная и полная документация сопровождает разработку ПО от появления идеи до выпуска конечного продукта. Мы не будем брать во внимание те случаи, когда подобная документация игнорируется по каким-либо причинам. Написание документации является обязательным критерием разработки и последующей поддержки проекта. По аналогии с чертежами здания — программная документация перед созданием проекта подскажет, где нужно забить сваи и где будут подводиться все коммуникации, а после создания проекта скажет — где есть несущие стены и какой модуль программы лучше не трогать, чтобы избежать большой беды.

Помимо формальных документов — таких как ТЗ, проект системы, тест-план программы и пр., разработчику программы необходимо владеть различными условно-графическими обозначениями (УГО), описывающими тот или иной аспект системы. Таким аспектом системы могут являться: структурная диаграмма всей системы; последовательность действий алгоритма, представленная в графическом виде; описание бизнес-процессов, с которыми должна работать разрабатываемая система. Наиболее часто встречаемыми на сегодняшний день нотациями, использующими УГО для концептуального описания того или иного аспекта системы, являются: нотации класса IDEF, нотации UML, ЕСПД (ГОСТ 19.XXX-XX). Их поверхностному описанию будет посвящена глава.



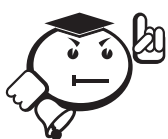
.....  
Следует отметить, что полное описание нотаций невозможно уложить в столь короткое пособие, поэтому читателю *следует*, с использованием приведённой в конце главы литературы, *изучить представленные нотации более подробно*.  
.....

## 6.1 Описание IDEF



.....  
*IDEF* (сокр. *Integration Definition Methodology* — Объединение Методологических Понятий).  
 .....

Семейство совместно используемых методов для решения задач моделирования сложных систем, позволяет отображать и анализировать модели деятельности широкого спектра сложных систем в различных разрезах.



.....  
 IDEF-технология используется, начиная с конца 1980-х годов. Министерство обороны США является основным пользователем данной технологии. Это обусловлено тем, что ВВС США в рамках программы компьютеризации промышленности ICAM (Integrated Computer-Aided Manufacturing) разработал стандарты IDEF.  
 .....

На данный момент существует 15 IDEF (0–14) стандартов, каждый предназначен для моделирования определённого бизнес-процесса. Читатель может спросить, а как же связаны бизнес-процессы и разработка ПО. Прямым образом! Разработка любого программного продукта направлена на решение задачи автоматизации того или иного бизнес-процесса. Поэтому использование определённой нотации для представления такого процесса с помощью УГО является необходимым для разработки программной системы.

Наиболее часто применяемыми при разработке ПО являются нотации: IDEF0 — методология функционального моделирования, используемая для описания статической функциональности разрабатываемой системы, IDEF3 — методология документирования процессов, происходящих в системе, которая используется, например, при описании работы модулей программы в зависимости от тех или иных условий.

### 6.1.1 IDEF0

Исторически, IDEF0 как стандарт был разработан в 1981 г. в рамках указанной ранее программы ВВС США. Мотивацией к разработке стандарта послужила необходимость в разработке новых методов анализа процессов взаимодействия в промышленных системах. Одним из важных требований к новому стандарту была возможность обеспечения работы в рамках «аналитик-специалист», то есть обеспечение групповой работы над разрабатываемым проектом.

В основе графического языка IDEF0 лежат четыре основных понятия: функциональный блок (activity box), интерфейсная дуга (arrow), декомпозиция (decomposition), точка зрения (view point). Обо всём по порядку.



.....  
**Функциональный блок (ФБ)** — графически обозначается в виде прямоугольника и олицетворяет собой некоторую конкретную функцию в рамках рассматриваемой системы.  
 .....

По требованиям стандарта название каждого ФБ должно быть сформулировано в глагольном наклонении (например, «производить услуги», а не «производство услуг»).

Каждая из четырёх сторон ФБ имеет своё определённое значение (роль) (см. рис. 6.1).

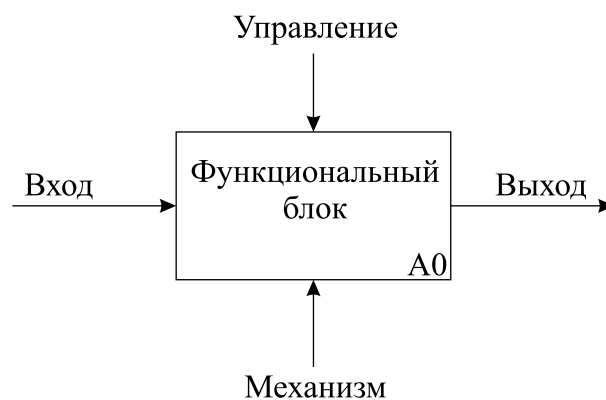


Рис. 6.1 – Функциональный блок

*Интерфейсная дуга (ИД)* — это информационные потоки, возникающие в системе, как видно на рисунке выше, ФБ уже связан четырьмя ИД.

УГО интерфейсной дуги является однонаправленная стрелка. Каждая ИД должна иметь собственное обозначение, сформулированное, по требованию стандарта, в виде существительного. В зависимости от того, к какой из сторон подходит интерфейсная дуга, она называется либо «входящей», «исходящей», либо «управляющей». Кроме того, началом и концом каждой функциональной дуги могут быть только ФБ, при этом «источником» может быть только выходная сторона блока, а «приёмником» любая из трёх оставшихся.

Также любой ФБ по требованиям стандарта должен иметь по крайней мере одну управляющую ИД и одну исходящую. Это и понятно — каждый процесс должен происходить по каким-то правилам (отображаемым управляющей дугой) и должен выдавать некоторый результат (выходящая дуга), иначе его рассмотрение не имеет никакого смысла. Рассмотрите пример, представленный ниже (рис. 6.2).

Разработка ПО может происходить под несколькими управляющими воздействиями (ТЗ, Проект системы, Стандарты кодирования и документации, Тестовый план и пр.) и с помощью нескольких механизмов (команды разработки и ПО, используемого для разработки). Следует отметить, что представлен пример идеальной разработки, когда все документы уже сгенерированы снаружи и команде разработки необходимо только выполнить кодирование.

Третьим основным понятием является декомпозиция как средство уменьшения сложности разрабатываемой системы. Любой сложный процесс для улучшения его

понимания необходимо разбить на более элементарные, атомарные процессы. Так и в IDEF0 каждый из функциональных блоков может уточняться и разбиваться на более элементарные блоки. Пример декомпозиции приведён ниже (рис. 6.3).

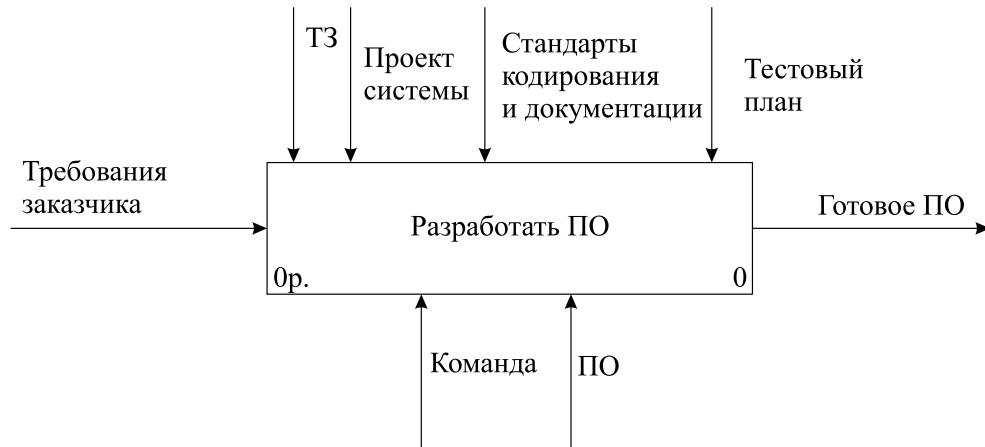


Рис. 6.2 – Пример верхнего уровня ФБ по разработке ПО

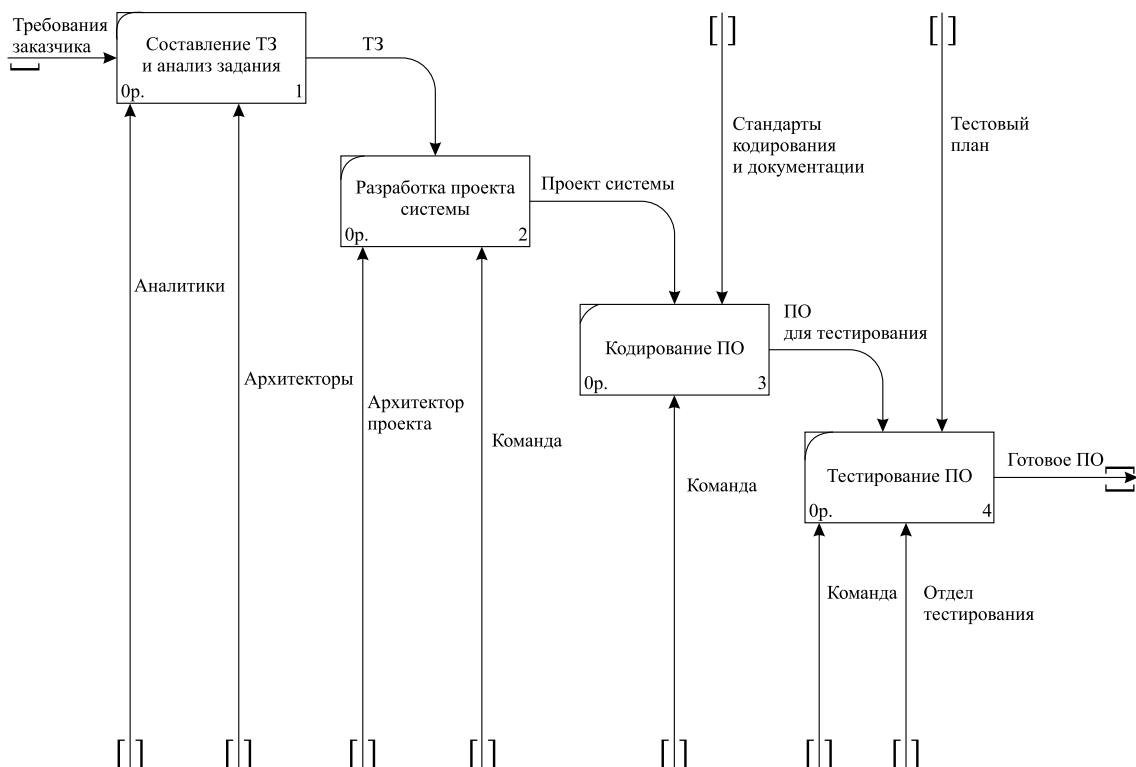


Рис. 6.3 – Декомпозиция ФБ разработки ПО

На примере показана декомпозиция ФБ, приведённого на рисунке 6.2. Читатель может увидеть, что ФБ выше не совсем соответствует его декомпозиции, и он окажется прав, т. к. представленный пример более реально описывает процесс разработки, т. к. ТЗ и проект системы приходится генерировать в процессе разработки ПО, поэтому пример моделирует такой процесс. Помимо этого, в примере декомпозиции команда разработки разбита на конкретные роли.

Диаграмма самого высокого уровня декомпозиции называется контекстной диаграммой и обозначается идентификатором «А-0».

В пояснительном тексте к контекстной диаграмме должна быть указана цель построения диаграммы в виде краткого описания и зафиксирована *точка зрения*. Определение и формализация цели разработки IDEF0 — модели является крайне важным моментом, т. к. цель определяет соответствующую область в исследуемой системе, на которой необходимо фокусироваться в первую очередь.

Точка зрения определяет основное направление развития модели и уровень необходимой детализации. Фактически мы описываем, насколько абстрактно представлена разрабатываемая система и для кого предназначена эта абстракция. Например, системному архитектору программы не важно полное описание процессов, происходящих в системе, для него необходимо поверхностное описание, позволяющее разрабатывать программный каркас системы. Для разработчиков конкретного модуля, наоборот, необходимо детальное описание определённого модуля, однако поверхностное описание системы уже не будет его интересовать.

Для упрощения разработки диаграмм по IDEF0-стандарту используются следующие ограничения:

- ограничения количества ФБ на диаграмме тремя-шестью. Верхний предел заставляет разработчика использовать различные иерархические уровни при описании системы, нижний предел гарантирует достаточную детальность разрабатываемой схемы;
- ограничение количества подходящих к одному ФБ (выходящих из ФБ) ИД четырьмя.

Следование этим ограничением является не обязательным, однако эти правила были получены опытным путём, поэтому их использование может упростить описание сложных систем.

Более подробное описание стандарта IDEF0 приведено в [43, 44]. Описание и пример использования CASE средств для создания диаграмм приведены в [45].

### 6.1.2 IDEF3

IDEF3 является стандартом документирования технологических процессов, происходящих на предприятии, и предоставляет инструментарий для наглядного исследования и моделирования их сценариев. В случае разработки ПО IDEF3 используется для описания динамических процессов, происходящих в системе. Сценарием называется описание последовательности изменений свойств объекта, в рамках рассматриваемого процесса (например, описание последовательности этапов модификации базы данных, изменение её свойств после прохождения каждого этапа).

В IDEF3 существует два типа диаграмм, описывающих два аспекта одного и того же процесса: диаграммы Описания Последовательности Этапов Процесса (Process Flow Description Diagrams, PFDD); диаграммы Состояния Объекта в Процессе его Трансформаций (Object State Transition Network, OSTN).

На диаграмме присутствуют УГО трёх видов: функциональные элементы, ФЭ (Unit of Behavior, UOB), линии, перекрёстки (Junction).




Функциональные элементы или элементы поведения обозначают событие, стадию процесса или принятие решения. Каждый ФЭ имеет своё имя, отображаемое в глагольном наклонении, и уникальный номер.

Линии бывают трёх видов:

- Старшая — сплошная линия, связывающая ФЭ.
- Отношения — пунктирная линия, используемая для изображения связей между ФЭ.
- Потoki объектов — стрелка с двумя наконечниками используется для описания того факта, что объект (деталь) используется в двух или более единицах работы.

Перекрёстки бывают 5 типов, все они приведены в таблице 6.1.

Таблица 6.1 – Описание перекрёстков, используемых в IDEF3

Обозначение	Наименование	Смысл в случае слияния стрелок (Fan-in Junction)	Смысл в случае разветвления стрелок (Fan-out Junction)
	Асинхронное И	Все предшествующие процессы должны быть завершены	Все следующие процессы должны быть запущены
	Синхронное И	Все предшествующие процессы завершены одновременно	Все следующие процессы запускаются одновременно
	Асинхронное ИЛИ	Один или несколько предшествующих процессов должны быть завершены	Один или несколько следующих процессов должны быть запущены
	Синхронное ИЛИ	Один или несколько предшествующих процессов завершаются одновременно	Один или несколько следующих процессов запускаются одновременно
	XOR (Исключающее ИЛИ)	Только один предшествующий процесс завершен	Только один следующий процесс запускается

На рисунке 6.4 приведены примеры IDEF3 диаграммы PFDD.

Сценарий, используемый на диаграммах, можно описать в следующем виде: команде разработки приходит сформулированный проект системы, с помощью которого программа поочередно разрабатывает архитектуру, модули и классы ПО, после этого необходимо выполнить тестирование системы. При этом, в идеальном случае, тестирование необходимо выполнять в следующем порядке: модульное, регрессивное, интеграционное, системное. Но, как описано раньше, это идеальный случай, обычно тестирование проводится в том порядке, в котором готовы необходимые для тестирования сущности: классы, модули, архитектура. После тестирования возможно следующее развитие событий: либо все тесты будут выполнены и тогда ПО отдаётся заказчику, либо один или несколько тестов не будут выполнены и необходимо будет вернуться к одному из предыдущих этапов разработки.

Более подробное описание стандарта IDEF3 приведено в [46]. Описание и пример использования CASE средств для создания диаграмм приведены в [45].

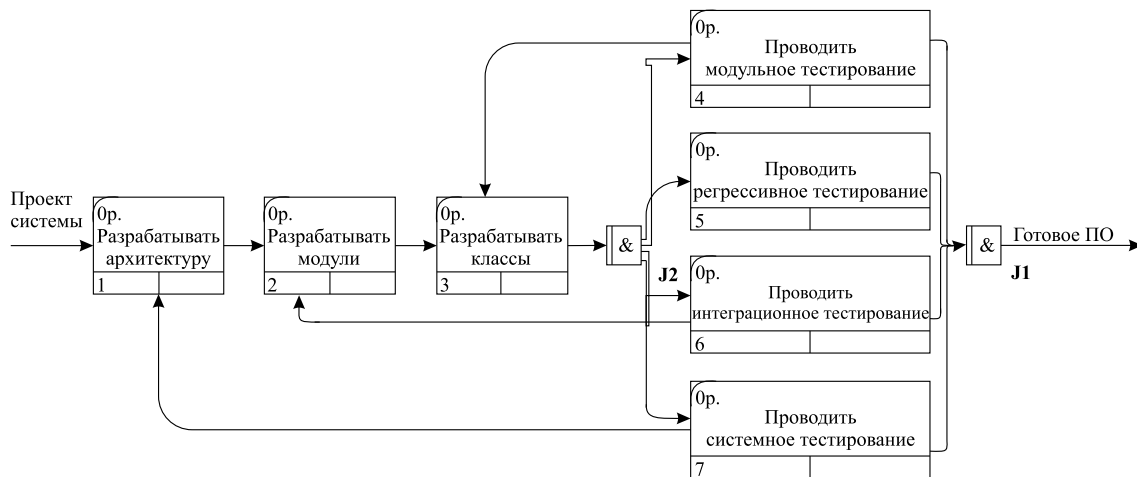


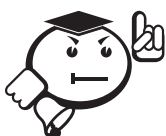
Рис. 6.4 – Пример PFDD диаграммы IDEF3

## 6.2 Unified Modeling Language

В этой главе рассмотрена нотация, которая в отличие от IDEF создавалась изначально для моделирования объектно-ориентированных (ОО) систем (глава 6). Сперва небольшой экскурс в историю. ОО языки моделирования появились в 1980-х годах, практически сразу после появления ОО языков программирования. Появление таких языков было направлено на уменьшение сложности анализа и проектирования ОО систем. В период с 1989 по 1994 год число ОО методов возросло с десяти до более чем пятидесяти. Естественно такое разнообразие методов и отсутствие стандартизации только вредили разработчикам.

В результате появилось новое поколение методов, среди которых особое значение приобрели метод Буча, OOSE (Object-Oriented Software Engineering), разработанный Якобсоном, и OMT (Object Modeling Technique), разработанный Рамбо. Каждый из представленных методов имел как свои достоинства, так и недостатки. Критическая масса новых идей накопилась к 1990-м годам, поэтому Рамбо, Якобсон и Буч решили унифицировать имеющиеся методики и создать один ОО язык моделирования. Перед собой они ставили три главные цели:

1. Моделировать системы целиком, от концепции до исполняемых компонентов, с помощью ОО методов.
2. Решить проблему масштабируемости, которая присуща сложным, критически важным системам.
3. Создать язык моделирования, который может быть использован не только людьми, но и компьютерами.



.....  
 Официальное создание UML началось в октябре 1994 года, и уже в 1995 свет увидела первая версия 0.8 языка UML. После нескольких ревизий в 1997 году был создан консорциум UML, включавший следующие корпорации: Hewlett-Packard, I-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Ratio-



nal, Texas Instruments и Unisys. UML оказался хорошо определённым, выразительным, мощным языком, применимым для решения большого количества разнообразных задач. В январе 1997 UML 1.0 был представлен перед консорциумом OMG (Object Management Group). После представления языка на консорциуме было принято решение о большей стандартизации и формализации языка. Пересмотренная версия UML 1.1 была представлена на рассмотрение OMG в июле 1997 года, а 14 ноября 1997 года она была взята за стандарт всеми членами консорциума.

С 1997 по 2003 гг. язык UML получил бурное развитие, и были представлены несколько новых версий языка, однако в 2005 году была утверждена последняя версия языка UML 2.0, которая на данный момент, с небольшими модификациями (стандарт 2.2), является стандартом индустрии разработки ПО.

.....

Полную спецификацию UML можно найти на сайте консорциума OMG [47]. UML — это результат работы множества специалистов и осмысления опыта их предыдущих разработок. С учётом всех научных исследований и рабочей практики можно сказать, что UML — это верхушка «айсберга», который вобрал в себя весь предыдущий опыт.

В UML существует порядка 14 типов диаграмм, которые в той или иной степени описывают поведение моделируемой системы (рис. 6.5).

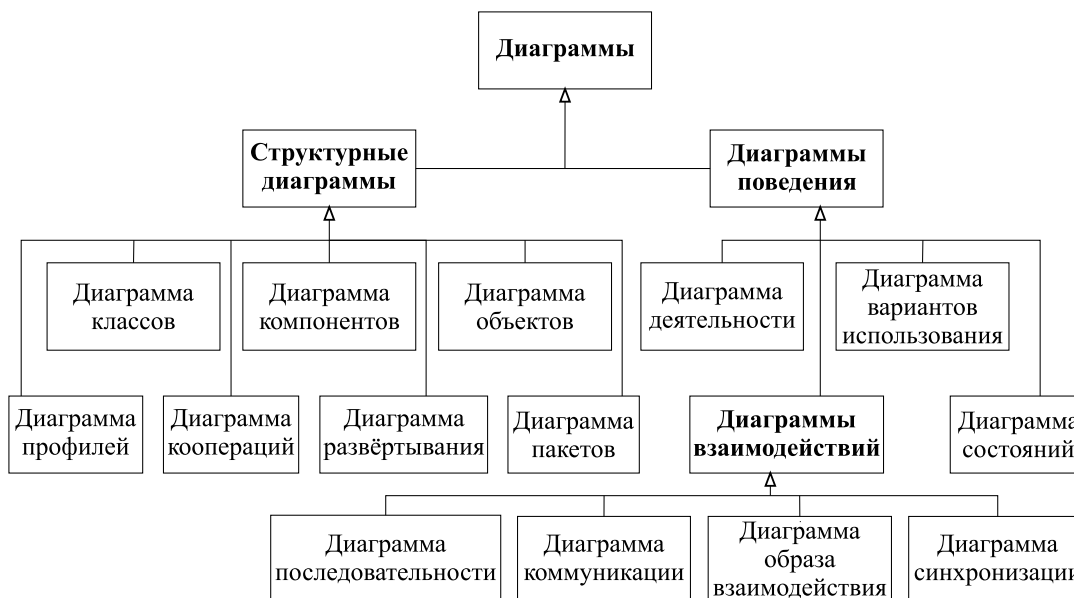


Рис. 6.5 – Виды диаграмм в стандарте UML

Рассмотреть с необходимой для практики скрупулёзностью каждый вид диаграмм не является целью данного пособия, наоборот, ниже будут представлены наиболее используемые виды диаграмм и некоторые детали, которые помогут читателю лучше ориентироваться в их использовании на практике. Ниже будут рассмотрены диаграммы деятельности (activity diagram), диаграммы вариантов ис-

пользования (use case diagram), диаграммы классов (class diagram), диаграммы пакетов (package diagram). Для более подробного ознакомления с предметом можно использовать следующие источники [48, 49]. С точки зрения авторов пособия, начинающим лучше воспользоваться относительно небольшой книгой Фаулера [49], а если читателю нужны глубокие знания в UML, тогда лучше воспользоваться источником [48].

### 6.2.1 Диаграмма деятельности



.....  
*Диаграммы деятельности* — это один из пяти видов диаграмм, применяемых в UML для моделирования динамических аспектов систем.  
 .....

По сути, диаграмма деятельности представляет собой блок-схему, которая показывает, как поток управления переходит от одной деятельности к другой. В отличие от традиционной блок-схемы диаграмма деятельности показывает параллелизм так же хорошо, как и ветвление потока управления. Более подробное описание блок-схем и их сравнение с диаграммами деятельности будет дано в пункте 6.3. При моделировании деятельности, под деятельностью будет пониматься структурированное описание текущего поведения. Диаграммы деятельности достаточно удобно использовать для описания используемого в программе алгоритма.

На рисунке 6.6 приведён пример диаграммы деятельности. На основе приведённого примера будут описаны основные сущности UML-диаграмм деятельности.

#### **Основные понятия.**

Деятельность (activity) — это выполняющийся в данный момент неатомарный набор действий внутри машины состояний (автомата). Действия заключаются в вызове другой операции, посылке сигнала, создании или уничтожении объекта либо в выполнении простых вычислений.

Каждая деятельность на диаграмме должна начинаться с *инициализации* и заканчиваться *завершением*. При этом стандарт UML строго не регламентирует количество этих сущностей, т. е. у алгоритма теоретически может быть как несколько начал, так и несколько завершений.

Атомарные элементы внутри машины состояний называются *действиями*. Следует заметить, что UML не предусматривает языка для подобных выражений. Абстрактно можно использовать структурированный текст, а более конкретно — синтаксис и семантику определённого языка программирования. Действия не могут быть подвергнуты декомпозиции.

*Узел деятельности* — это организационная единица деятельности. Вообще он представляет собой вложенную группу действий или других вложенных узлов. Действие можно представить как частный случай узла деятельности.

Когда некоторое действие или узел деятельности завершает выполнение, фокус управления с помощью *потоков управления* переходит к следующему действию.

Различные условные операторы представляются в виде *ветвлений*, на которых подписаны условия перехода к тому или иному действию.

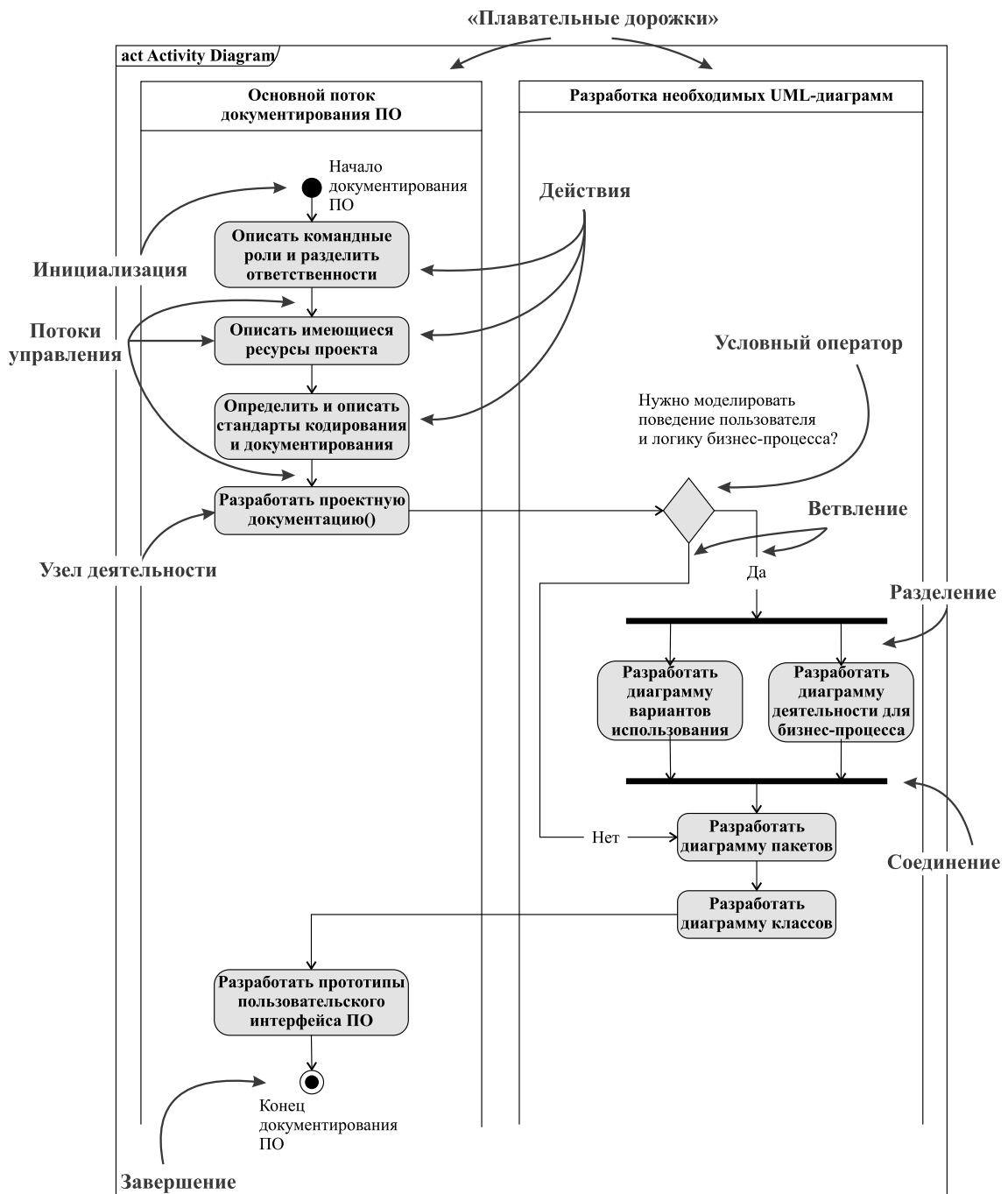


Рис. 6.6 – Пример диаграммы деятельности

Для описания синхронных процессов используются такие сущности, как *разделение* и *соединение*. Эти сущности описывают, что все потоки управления начинаются одновременно *после* разделения и все потоки управления завершаются одновременно *перед* соединением.

Для более удобной декомпозиции по определённому признаку обычно используются так называемые «плавательные дорожки» (swimlines).

При моделировании динамических аспектов системы диаграммы деятельности обычно используются двумя способами:

1. Для моделирования потока работ. Внимание уделяется тому, как выглядит деятельность с точки зрения действующих лиц, взаимодействующих с системой.
2. Для моделирования операции. В этом случае диаграммы деятельности выступают в качестве схем, моделирующих подробности вычислительного процесса (алгоритма выполнения).

## 6.2.2 Диаграмма вариантов использования

Ни одна система не существует в изоляции: она взаимодействует с действующими лицами (людьми или системами), которые используют её для достижения некоторой цели, ожидая от неё определённого поведения. Вариант использования (ВИ) специфицирует это ожидаемое поведение субъекта (системы или её части), — он описывает последовательности действий, включая их варианты, которые субъект осуществляет для достижения действующим лицом определённого результата.

ВИ служат для описания взаимодействия системы с одним или несколькими действующими лицами. Фактически диаграмму ВИ удобно применять при анализе требований к функциям, доступным для пользователей. Одной из важных особенностей таких диаграмм является то, что ВИ не специфицируют конкретный способ реализации той или иной функции, вы только описываете в общем виде ВИ вашей системой с точки зрения действующего лица. Такой подход позволяет не фокусироваться на деталях реализации и абстрагировать функциональность системы от всех технических особенностей системы.



.....  
**Вариант использования (use case)** — это описание множества последовательных действий (включая вариации), которые выполняются некоторым субъектом с целью получения результата, значимого для некоторого действующего лица.  
 .....

ВИ предполагает взаимодействие действующих лиц и системы или другого объекта. Действующее лицо представляет собой логически связанное множество ролей, которые играют пользователи системы во время взаимодействия с ней. Действующими лицами могут быть как люди, так и автоматизированные системы. Пример УГО действующего лица и ВИ приведён на рисунке 6.7.

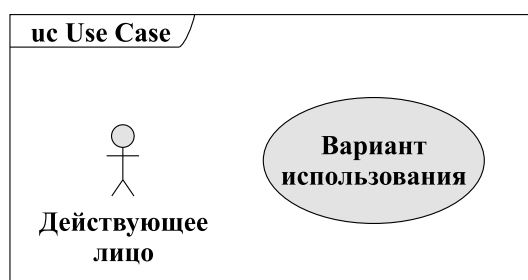


Рис. 6.7 – Пример действующего лица и варианта использования

Следует заметить, что не существует формальной спецификации, которая ограничивает название ВИ, т. е. теоретически можно использовать длинные конструкции, однако следует предостеречь читателя от этой возможности, т. к. короткие и лаконично сформулированные варианты использования могут помочь избежать разного толкования и сделают вашу диаграмму более удобной для восприятия и, как следствие, более полезной. Единственное, чего требует стандарт от именования ВИ, — это использование глагольных конструкций в действительном залоге, например: ввести параметры элементов, считать файл, отправить сообщение пользователям и т. п.

При моделировании ВИ в UML каждый из них должен предоставлять некоторое отдельное и идентифицируемое поведение системы или её части. Хорошо структурированный ВИ обладает следующими свойствами:

- именуется простое, идентифицируемое и атомарное (в той степени, в которой это нужно) поведение системы или её части;
- выделяет общее поведение, извлекая его из других ВИ;
- выделяет вариации, помещая некоторое поведение в другие ВИ, расширяющие его;
- описывает поток событий так, чтобы сделать его понятным читателю;
- описывается минимальным набором сценариев, специфицирующих его основную семантику и семантику вариаций.

Основные связи между вариантами использования и другие базовые понятия будут рассмотрены на примере, приведённом на рисунке 6.8.

*Связь включения (include)* — обозначает обязательность выполнения включаемого ВИ. Если один вариант использования включает другой вариант использования, то этот вариант использования не будет выполнен до тех пор, пока не будет выполнен включаемый ВИ. Графически обозначается пунктирной однонаправленной стрелочкой к *включаемому* ВИ с подписью «include».

*Связь расширения (extend)* — обозначает связь типа «конкретизация при условии». Базовый ВИ способен существовать отдельно, но при некоторых условиях его поведение может быть расширено поведением другого ВИ. Базовый ВИ можно расширить только вызовом из определённой точки, — так называемой *точки расширения (extension point)*. Чтобы наглядно представить ситуацию, вообразите, что расширяющий ВИ «вталкивает» поведение в базовый. Связь расширения используется для моделирования необязательных с точки зрения пользователей ВИ, отделяя таким образом обязательные ВИ от необязательных. Графически обозначается пунктирной однонаправленной стрелочкой к *расширяемому* ВИ с подписью «extend».

*Связь обобщения (generalize)* — обозначает связь типа «общее-частное». Используется для описания взаимодействия с общим ВИ, который может быть конкретизирован в зависимости от необходимости. Графически обозначается однонаправленной цельной стрелочкой с незакрашенным треугольником на конце, направленной от частного ВИ к общему.

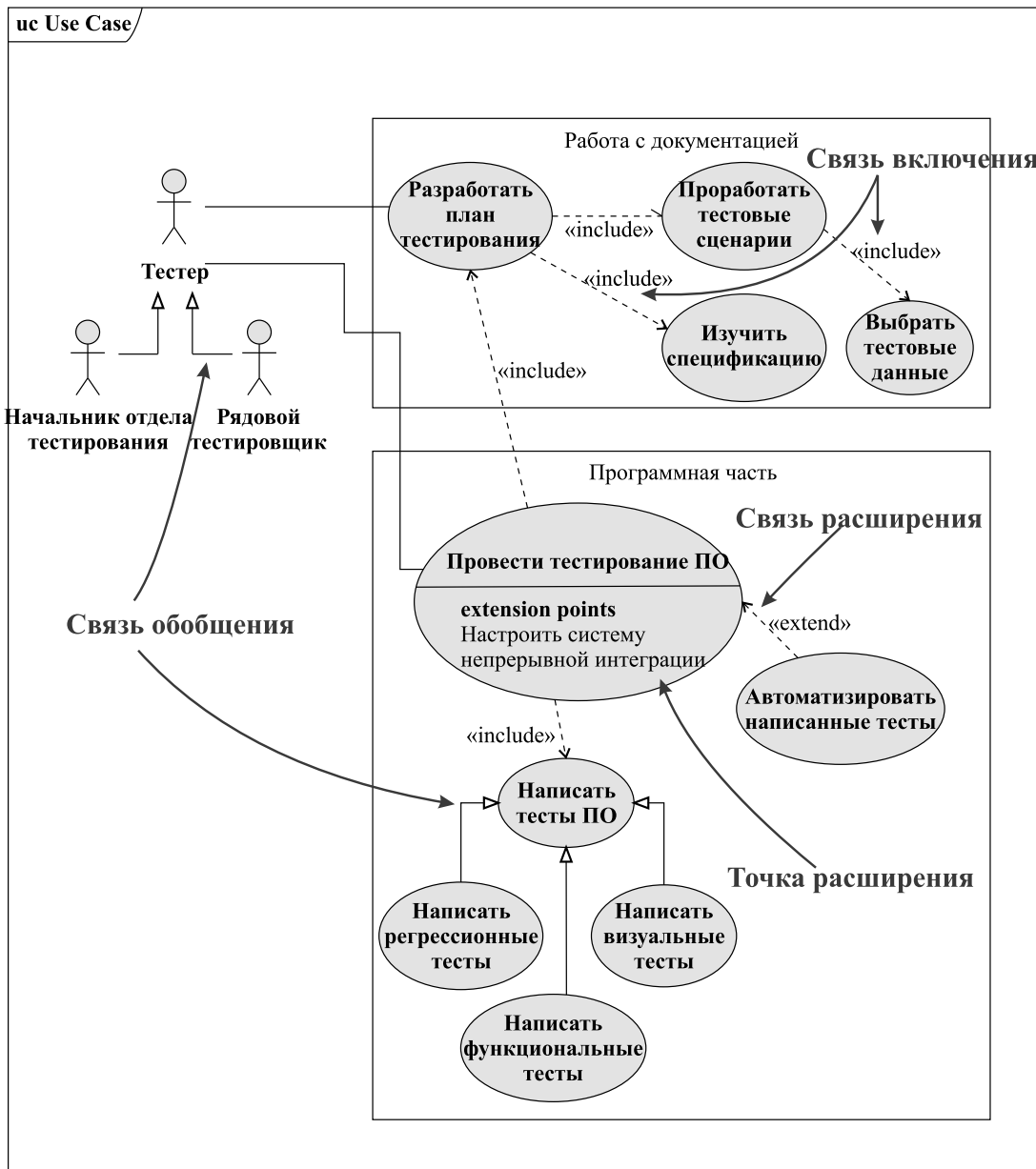


Рис. 6.8 – Пример диаграммы деятельности



.....  
 Следует отметить, что диаграммы ВИ достаточно слабо формализованы. Такая слабая формализация связана с использованием естественного языка для описания самих ВИ. Однако использование именно естественного языка позволяет наиболее полно описать ВИ разрабатываемой системы и учесть все пожелания заказчика.

При построении диаграммы ВИ следует обратить внимание на то, что ко всем вариантам использования должна быть возможность перейти с помощью связей. Один ВИ должен либо включать другой, либо расширяться или обобщаться другим ВИ. При этом к каждому ВИ должно иметь доступ хотя бы одно действующее лицо.

.....

## 6.2.3 Диаграмма классов



.....  
**Классы** — наиболее важные строительные блоки любой объектно-ориентированной системы.  
 .....

Класс — это описание множества объектов с одинаковыми атрибутами, операциями, связями и семантикой. Особенности конструирования классов и наполнения их поведением и смыслом читатель может прочитать в специализированной литературе, например в [50]. В данной главе класс рассматривается как структурная единица любой объектно-ориентированной программы. Так же, как и при любом создании строительного сооружения, необходимо иметь чертёж, где будет достаточно подробно описано, где сваи прикреплены к каркасу, в каких местах заложена дополнительная прочность и какие из стен являются в конструкции несущими. Таким чертежом при разработке ПО может выступать именно диаграмма классов.

УГО класса с основными обозначениями приведено на рисунке 6.9.

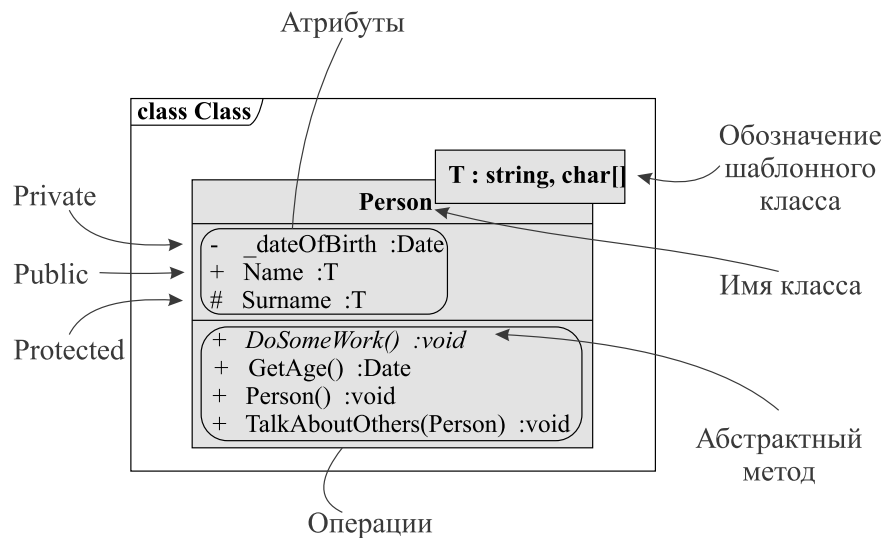


Рис. 6.9 – УГО класса в UML нотации

Ниже опишем основные понятия, используемые при обозначении класса.

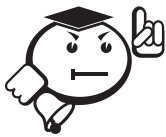
**Имя класса** — текстовая строка, обозначающая название класса. Отдельное имя является именем самого класса, а префикс в имени обозначает имя пакета, в котором находится класс (о диаграмме пакетов речь пойдёт ниже). При необходимости убрать детали реализации класс можно изображать без атрибутов и операций, оставив только одно имя.

**Атрибут (поле)** — это именованное свойство класса. Класс может иметь любое количество атрибутов или не иметь ни одного. Атрибут представляет некоторое свойство моделируемой сущности, которой обладают все объекты данного класса. Атрибут можно уточнять, указывая его класс и начальное значение по умолчанию.

*Операции (методы)* — это реализация поведения, которым обладает объект моделируемого класса. Так же, как и в случае с атрибутами, класс может иметь любое количество операций. Помимо именованной операции в классе можно также описать сигнатуру операции: типы входных данных и тип возвращаемого значения.

*Модификаторы доступа* — обозначения доступности того или иного атрибута или операции для другого класса. В объектно-ориентированных языках существуют несколько модификаторов доступа, для каждого из них есть своё символическое обозначение:

- Private — символическое обозначение — показывает, что атрибут или операция доступна только внутри класса. Обычно такие атрибуты или операции не выносятся на диаграммы классов для сохранения *инкапсуляции*.
- Public — символическое обозначение +, показывает, что атрибут или операция доступна для всех классов.
- Protected — символическое обозначение #, показывает, что атрибут или операция доступна только из этого класса или из классов наследников.



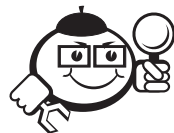
.....  
 Статические поля и методы класса изображаются с помощью подчёркивания. Статические классы изображаются с помощью подчёркивания имени класса. Что означает static в каждом из языков программирования, читатель должен узнать самостоятельно.

Абстрактные методы класса обозначаются с помощью курсива. Абстрактные классы обозначаются курсивом в имени класса.

Шаблонные классы обозначаются с помощью квадрата в правом верхнем углу класса.

.....

Как и в любой системе, между классами существуют взаимодействия, описываемые определёнными связями на диаграмме. Моделирование подобных взаимодействий позволяет получить наиболее полное представление о создаваемой системе. Все существующие типы связей будут разобраны на примере, приведённом на рисунке 6.10.



## Пример

.....

Немного слов о примере: этот пример моделирует взаимодействие студентов и профессора в рамках одного занятия. Помимо этого, в системе присутствуют такие сущности, как дата (Date), описывающая необходимую дату, сущность «человек» (Person), которая описывает общие черты, присущие классам «профессор» и «студент», интерфейс IDrawable, добавляющий в любую сущность возможности по отрисовке. Сразу следует отметить, что расставлены не все возможные связи, для того, чтобы не засорять диаграмму, так, например, не стоит связь использования между классом *Человек* и датой, хотя такая связь должна быть. Также следует отметить связь агрегации между классом *Занятие* (Lesson) и классом *Че-*



овек (Person). Агрегация как связь хранения по ссылке подразумевает создание экземпляров класса *Человек* где-то в другом классе и передачу их в класс *Занятие* для использования и хранения по ссылке, без возможности удалить изначальный объект. Класс, порождающий экземпляры класса *Человек*, не показан на представленном примере, однако это может быть определённый управляющий класс (*Manager*), получающий эти экземпляры из БД, по сети или из пользовательского интерфейса напрямую. Связи композиции к классу *Date* показывают, что в композирующих классах жёстко хранятся экземпляры даты и если композирующие объекты будут удалены, то их объекты даты тоже будут удалены вслед за ними. Подробное описание каждого вида связи приведено ниже.

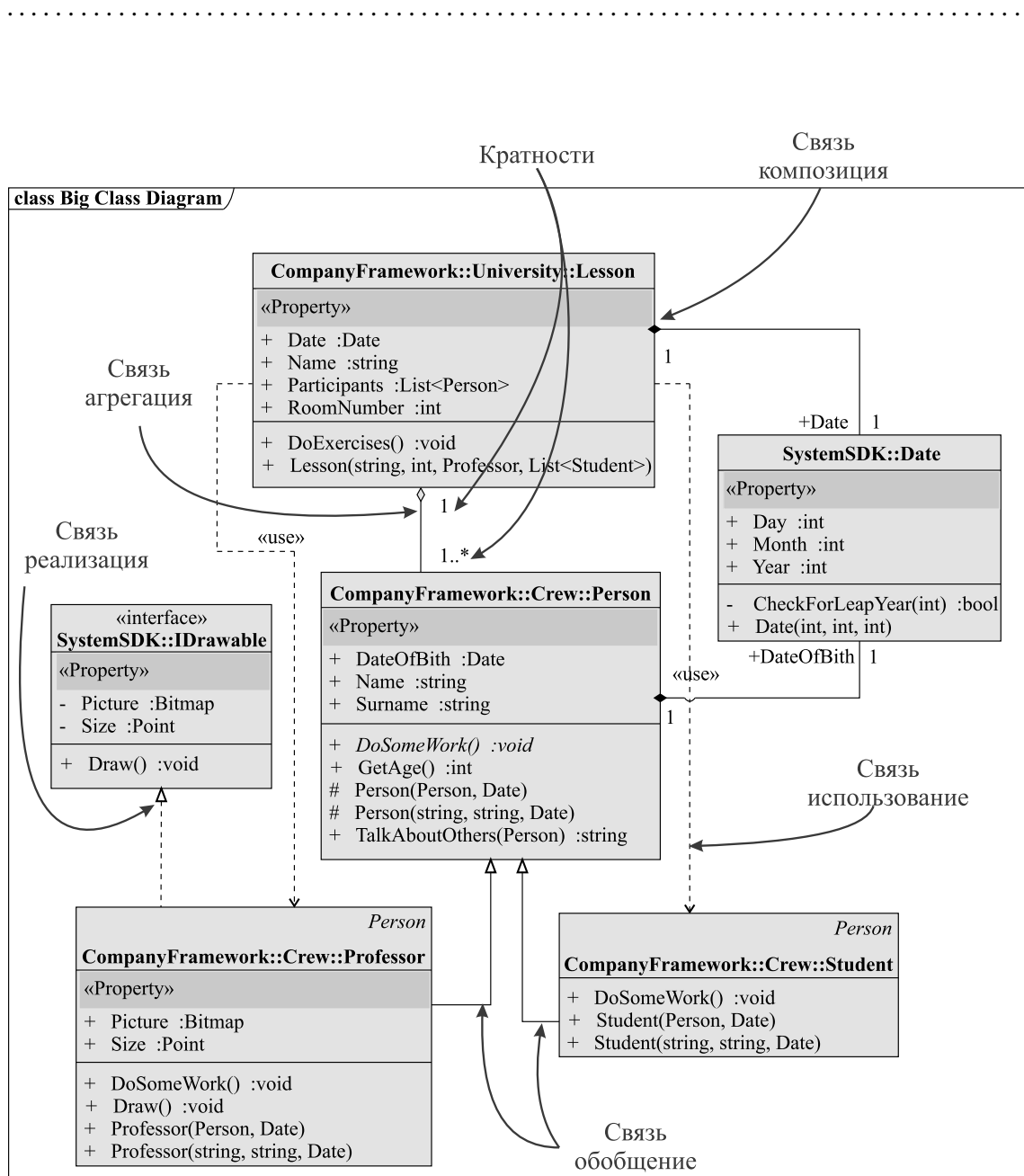


Рис. 6.10 – Пример диаграммы классов со связями

Связи, входящие в одну группу, называемую ассоциацией: агрегация и композиция. В общем виде ассоциация — это структурная связь, показывающая, что объекты одной сущности соединены с объектами другой. Ассоциация изображается сплошной линией. Ассоциация может быть однонаправленной, двунаправленной или не направленной, в соответствии с этим на связи ассоциация ставится одна стрелка, две стрелки в обе стороны или вообще не ставятся стрелки. Чаще всего ассоциация ставится на концептуальном уровне проектирования, когда разработчики до конца не знают, как будут связаны классы, но уже уверены, что они будут связаны как «часть-целое». Над связью ассоциации иногда ставится кратность, что это такое — описано ниже.

*Агрегация* — слабый вид ассоциации, обозначает связь обладания. То есть один объект класса обладает другим объектом класса, однако обладающий класс не создавал объект другого класса и, значит, не включает его как «часть-целое». Проверить наличие этой связи можно следующим мысленным действием, попробовать удалить агрегирующий объект класса, если при этом объект агрегируемого класса продолжит существовать, то это связь агрегации.



## Пример

Примеры агрегации в разных языках программирования: C# — хранение объекта по ссылке, т. е. передача в один объект из другого, что, соответственно, позволит существовать агрегируемому объекту после удаления агрегирующего, C++ — хранение объекта по указателю и умышленное игнорирование объекта (ов) при вызове деструктора класса, что приведёт к «зависанию» объекта в памяти и, при осознанном процессе, возможности использования этого объекта где-то ещё, при неосознанном процессе, приведёт к «утечке» памяти и трате большого количества времени на нахождение этой трудноуловимой ошибки.

УГО является цельная стрелочка или линия с пустым ромбом на стороне агрегирующего класса и стрелочкой или без неё, направленной к агрегируемому классу.

*Композиция* — строгая связь ассоциации, которая подразумевает хранение объектов одного класса в другом как «часть-целое». Такой вид связи можно проверить обратным способом, представленному выше. Если объект композиремого класса перестает существовать после удаления объекта композирющего класса, то это связь — композиция. Иными словами, можно сказать, что время жизни композиремого и композирющего классов совпадает, но не наоборот.

*Кратности (кардинальности)* — обозначают соотношения экземпляров одного класса к экземплярам другого класса. То есть сколько будет создано объектов одного класса для другого класса. Существуют следующие виды кратностей: единица (1), ноль или один (0..1), много (0..\*), один или несколько (1..\*). При этом стандарт жёстко не ограничивает использование кратностей, поэтому их можно уточнять в зависимости от моделируемой системы.

Для всех связей семейства ассоциаций следует применять следующие правила при проставлении:

- если в полях или свойствах класса есть переменные типа другого класса, значит, эти классы относятся по соотношению ассоциации;
- если в полях или свойствах класса присутствуют массивы, списки или другие количественные переменные типа другого класса, то, скорее всего, кратности будут следующие: для ассоциирующего класса — 1, для ассоциируемого класса 0..\*.

Следующим способом создания новых классов, на основе имеющихся, после ассоциации является обобщение. Связь *обобщение* — является связью, которая в объектно-ориентированных языках моделирует наследование, т. е. заимствование публичных и защищённых полей и методов класса-родителя классом-потомком. УГО связи обобщения является цельная стрелка с пустым треугольником на конце. Стрелка направляется от класса-наследника к классу-родителю.

В языках, где существует возможность использования таких программных сущностей, как интерфейсы (просьба не путать с интерфейсами классов, т. е. общедоступными полями и методами и графическими пользовательскими интерфейсами), используется ещё один вид связи — реализация. *Реализация* — это связь, схожая по типу со связью обобщение. Для того, чтобы прояснить использование реализации, следует дать общее определение интерфейсу, интерфейс — это программная сущность, которая описывает свойства объекта на определённом уровне абстракции предметной области, без его конкретизации, т. е. описания конкретной реализации поведения и свойств. Фактически интерфейс только описывает спецификацию этого объекта, т. е. используемые типы данных и типы возвращаемых значений.



## Пример

В качестве примеров интерфейсов в различных языках программирования можно привести интерфейсы в C# (interface) и абстрактные классы в C++ (класс, имеющий хоть один чисто виртуальный метод). Связь реализации показывает, что один класс реализует определённый интерфейс, т. е. получает некоторые свойства и поведение, свойственные определённому уровню абстракции. Реализация одним классом интерфейса говорит о необходимости конкретизации этих абстрактных свойств в реализующем классе, т. е. придании этим свойствам конкретики: назначение определённых свойств или описание реализации методов интерфейса.

УГО связи реализации является пунктирная стрелочка, направленная от класса к интерфейсу, с пустым треугольником на конце, как и у связи обобщение.

Самый слабый тип связи из представленных — это зависимость. *Зависимость* — это связь использования, указывающая, что изменение спецификации (публичных полей и методов) одной сущности, может повлиять на другие сущности, которые её используют. Направляется от зависимой сущности к классу, от которого зависит сущность.



## Пример

С наибольшей вероятностью можно сказать, что одна сущность использует другую, если используемая сущность является входным типом параметров или выходным типом методов класса, в котором она используется. Другими словами, если в сигнатуре методов в качестве входных параметров метода, определённого класса 1 используется другой класс 2, то класс 1 зависит от класса 2.

УГО зависимости является однонаправленная пунктирная стрелка, направленная к зависимому классу.

В стандарте UML существует много обозначений, уточняющих различные аспекты моделируемой системы с помощью диаграммы классов, читатель может с ними ознакомиться в представленной ранее литературе, т. к. учебного пособия для этого будет мало. Хотелось бы заметить, что иногда для описания системы нет необходимости в подробном описании каждого класса в системе, т. е. существует необходимость описать систему на самом верхнем уровне, что зачастую нужно для того, чтобы дать архитектору системы или пользователю API представление о компонентах ПО. В таком случае используется диаграмма пакетов.

### 6.2.4 Диаграмма пакетов

Визуализация, специфицирование, конструирование и документирование больших систем предполагают работу с множеством классов, интерфейсов, узлов, компонентов, диаграмм и других элементов. Масштабируя такие системы, вы столкнётесь с необходимостью организовывать эти сущности в крупные блоки. В языке UML для организации элементов модели в группы используют пакеты.

Пакет — это способ организации элементов модели в блоки, которыми можно распоряжаться как единым целым. Можно управлять видимостью элементов пакета, так что некоторые будут видны пользователю, а другие — скрыты. Кроме того, с помощью пакетов изображаются различные представления архитектуры системы.

Хорошо структурированный пакет объединяет семантически близкие элементы, которые имеют тенденцию изменяться совместно. Такие пакеты характеризуются слабой связностью и высокой согласованностью, причём доступ к содержимому пакета строго контролируется.

Как известно — любая система может декомпозироваться до элементарных единиц вниз и до самых крупных блоков вверх по иерархии. Такая декомпозиция позволяет бороться с огромной сложностью программной системы, скрывая реализацию в виде отдельных классов и показывая только необходимые пакеты в нужный момент времени.

Чтобы было более понятно, рассмотрим пример из жизни. Разрабатывая жилой дом, архитектор имеет укрупнённый чертёж дома, где показано, как у дома должны выглядеть основные конструкции: фундамент, несущие стены, внешний каркас дома, крыша и др. Фактически каждая такая сущность — это пакет, который в себе скрывает определённую часть реализации. Например, возьмём фундамент — как

только мы заходим в блок фундамент, то мы видим, как будут организованы внутренние помещения подвала, где в дом будут заводиться коммуникации (отопление, канализация, электропитание), где находятся наиболее нагруженные сваи, которые отвечают за устойчивость всего здания. Заходя в каждое из помещений (пакетов) подвала, мы уже видим конкретное устройство помещения, сантехнические трубы, провода, стены, двери и т. д. Таким образом, можно разложить каждую крупную сущность дома, пока не будет видна конкретная реализация. Пример диаграммы пакетов приведён на рисунке 6.11.

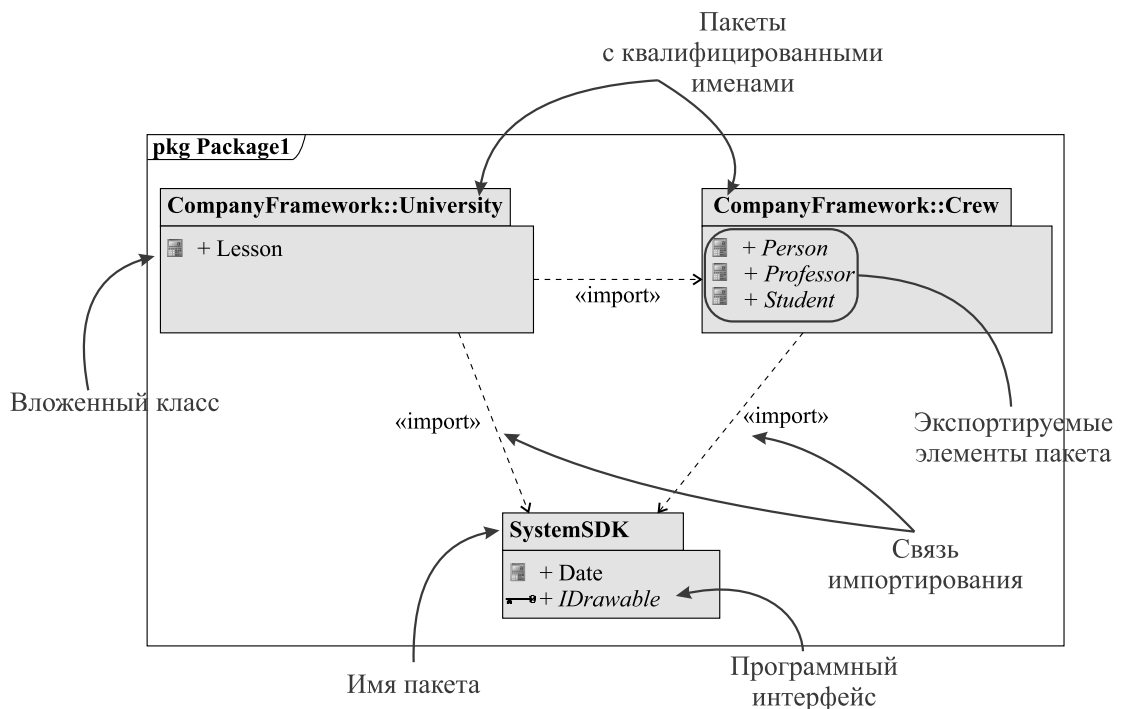


Рис. 6.11 – Пример диаграммы пакетов со связями

Пример выше моделирует то же самое, что и моделирует диаграмма классов выше (рис. 6.10). На данном примере можно увидеть, что классы сгруппированы в более крупные сущности (пакеты), описание которых на определённом уровне является достаточным для описания системы.

Пакет — представляет собой общий механизм организации элементов в группы. Изображается в виде папки с закладкой. Имя пакета указывается на папке, если содержимое последней не показано, а в противном случае — на закладке. У каждого пакета должно быть имя, отличающее его от других пакетов. Обычно в качестве имени используется простая текстовая строка. Квалифицированное имя предваряется именем пакета, включающего данный пакет, если такое вложение имеет место, такое разделение выполняется с помощью двойного двоеточия (::).

Иногда пакет может обладать другими пакетами или классами, тогда эти пакеты или классы рисуются напрямую в пакете. Считается, что эти элементы компонируются в пакете и, как следствие, удаляются после удаления главного пакета.

В качестве связей между пакетами используется *импортирование*, т. е. когда элементы одного пакета обращаются к доступным элементам другого пакета. От-

крытые элементы пакета называются *экспортируемыми*. Очень часто экспортируемыми частями пакетов являются интерфейсы.



.....  
 В качестве программного индикатора наличия пакета следует использовать так называемые пространства имён (namespace). Разделение программы на пространства имён также позволяет избежать большой сложности системы и сгруппировать классы в общие структурные элементы.  
 .....

### 6.3 Блок-схемы

Альтернативой диаграммам активностей UML являются блок-схемы, определенные в Единой системе программной документации (ЕСПД). ЕСПД — это набор государственных стандартов Российской Федерации, определяющих правила оформления программной документации. К данной группе относятся стандарты с классификационными номерами ГОСТ 19.XXX-XX.

Согласно ГОСТ 19.701-90 блок-схемы подразделяются на пять типов:

- 1) схемы данных;
- 2) схемы программ;
- 3) схемы работы системы;
- 4) схемы взаимодействия программ;
- 5) схемы ресурсов системы.

В данном разделе мы рассмотрим правила оформления только схем программ, предлагая читателю самостоятельно ознакомиться с остальными требованиями ЕСПД.

В отличие от диаграмм активностей UML блок-схемы являются более строгим видом диаграмм. В частности, это определяется:

- а) большим количеством возможных условных графических обозначений;
- б) более строгими требованиями к начертанию элементов схемы;
- в) строгим определением размеров элементов схемы.

С одной стороны, это усложняет разработку подобных схем. Однако на практике такие схемы более удобочитаемы и более информативны, так как стандарт обеспечивает аккуратное представление каждого процесса. При этом, если в диаграмме активностей мы можем нарисовать блок любого размера и разместить в нём текст любого объёма, то блок-схемы ЕСПД требуют одинакового размера для всех блоков процесса. Данный факт заставляет задумываться о содержательности текста и, самое главное, учит грамотно декомпозировать алгоритм на *одинаковые, соизмеримые по сложности* процессы.

#### **Условно-графические обозначения.**

В первую очередь перед проектированием алгоритма необходимо определить ширину  $a$  и высоту  $b$  блоков. Данные величины будут использоваться при начертании *всех* блоков схемы. Дополнительным ограничением на высоту является

требование кратности  $0.5a$ . Обычными значениями являются соотношения 5:3 или 5:2.5. Все последующие начертания условно-графических обозначений выполняются согласно выбранным вами значениям  $a$  и  $b$ .

Символ (см. рис. 6.12) отображает функцию обработки данных — выполнение определенной операции или групп операций. В качестве текста указывается выполняемая операция.

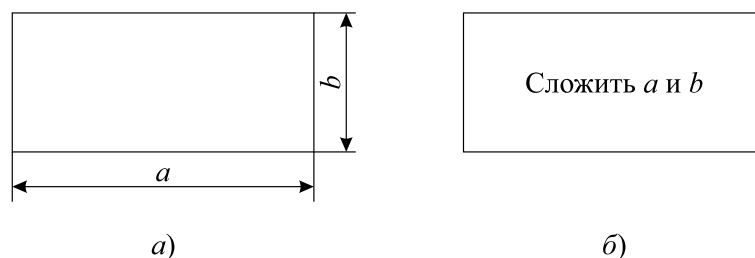


Рис. 6.12 – УГО символа процесса

Символ (см. рис. 6.13) отображает данные, при этом носитель данных не определен или не важен. В качестве текста указывается операция чтения или записи и перечень данных.

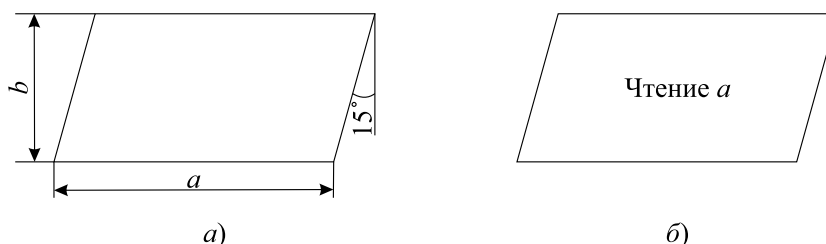


Рис. 6.13 – УГО символа данных

Данный символ используется для показа чтения или записи данных.

Символ (см. рис. 6.14) отображает условие или функцию переключательного типа, когда дальнейшее выполнение алгоритма выполняется по одному из возможных вариантов. Данный блок имеет строго один вход, обозначаемый стрелкой, и несколько альтернативных выходов, один и только один из которых может быть активизирован после вычисления условия, указанного в тексте символа. Каждый из выходов должен быть подписан некоторым значением — результатом условия, при котором будет активирован данный выход. Если из обозначения выходов очевидно условие, условие может не указываться в качестве текста внутри блока.

Символ (см. рис. 6.15) отображает поток данных или управления. Направление стрелки указывает, какому процессу передается управление. Направление стрелки может не указываться в случае передачи управления вниз или вправо, однако указание стрелки обязательно для входа решения.

Линии могут соединяться между собой. Если пересечение линий не предполагает соединения, пересечение обозначается дугой. В любом случае, в схеме следует избегать пересечения линий.

Направление линий не регламентируется, однако для удобочитаемости предпочтительны прямые линии под прямыми углами.

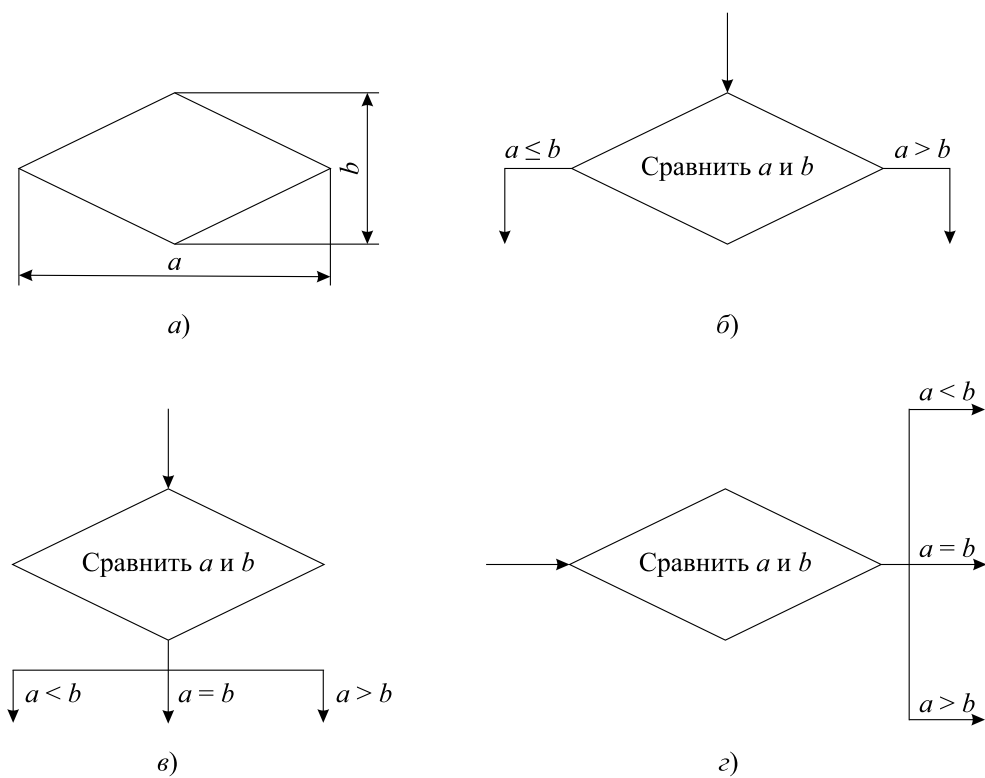


Рис. 6.14 – УГО различных видов условий

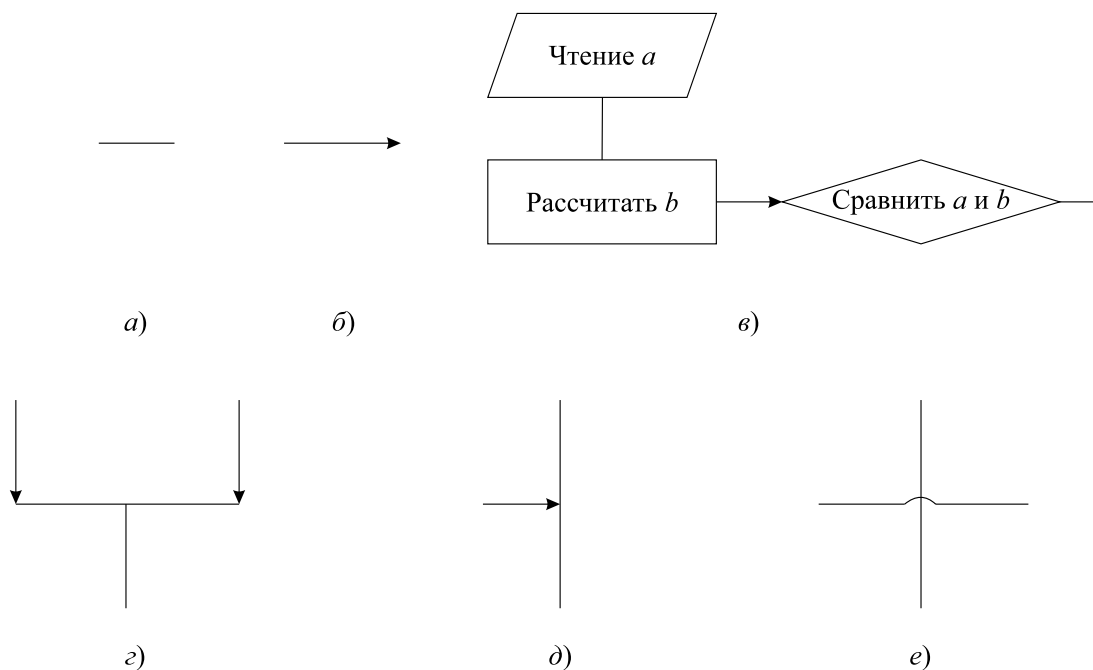


Рис. 6.15 – УГО потоков данных или управления

Символ (см. рис. 6.16) отображает синхронизацию двух или более параллельных операций. Данный символ используется, когда процессы могут выполняться параллельно либо нам не важен порядок их выполнения.



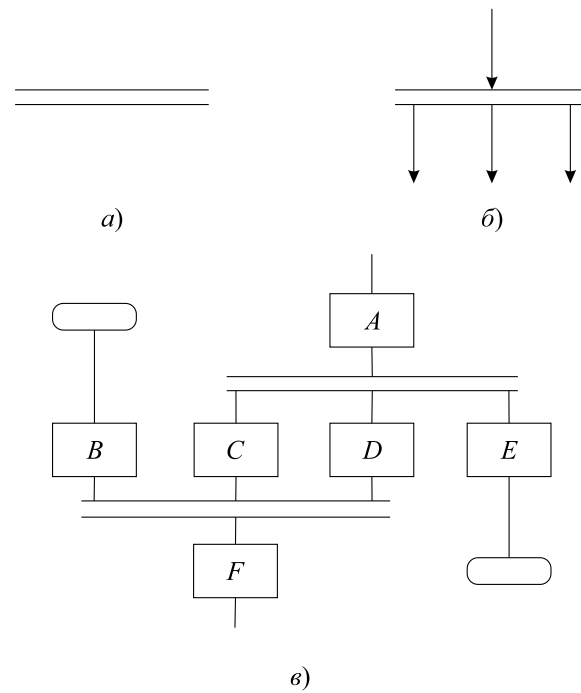


Рис. 6.16 – УГО символов параллельных операций

В случае входа двух параллельных процессов в блок параллельных действий выходной процесс не начнется, пока не выполнятся все входные процессы. Таким образом, на рисунке 6.16, в процессы *C*, *D* и *E* не начнутся, пока не завершится процесс *A*, а процесс *F* не начнется, пока не закончатся процессы *B*, *C* и *D*.

Символ (см. рис. 6.17), состоящий из двух частей, отображает начало и конец цикла. Обе части символа имеют один и тот же идентификатор. Условия для инициализации, приращения, завершения и т. д. помещаются внутри символа в начале или в конце в зависимости от расположения операции, проверяющей условие.

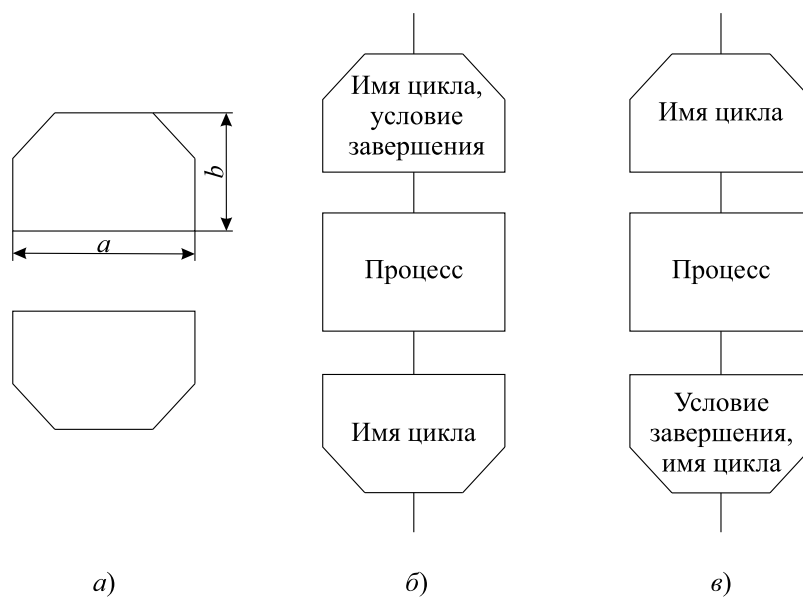


Рис. 6.17 – УГО начала и конца цикла

Сравнение UML-диаграммы деятельности и блок-схемы приведено на рисунке 6.18.

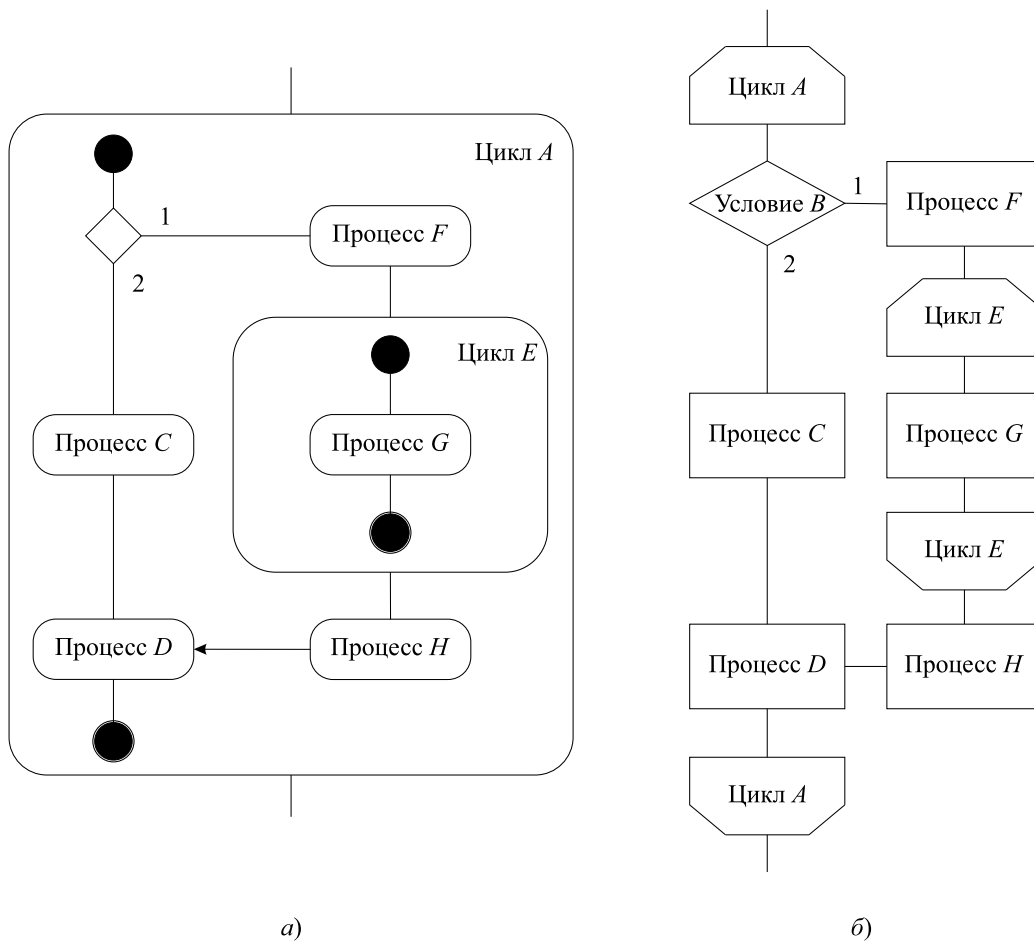


Рис. 6.18 – УГО сравнения циклов: а) в диаграммах деятельности, б) в блок-схемах

Символ (см. рис. 6.19) отображает predetermined процесс, состоящий из одной или нескольких операций, которые определены в другом месте (в подпрограмме, модуле). Используется для указания вызова функции или подпрограммы, представленной, например, на другой блок-схеме.

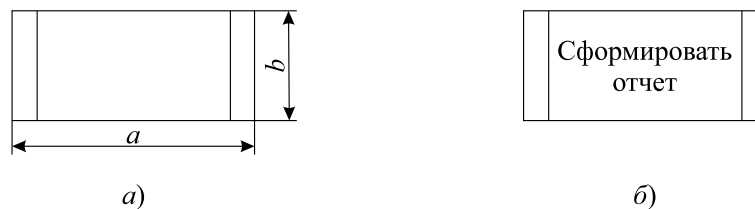


Рис. 6.19 – УГО символа predetermined процесса

Символ (см. рис. 6.20) отображает выход в часть схемы и вход из другой части этой схемы и используется для обрыва линии и продолжения её в другом месте. Соответствующие символы-соединители должны содержать одно и то же уникальное обозначение.

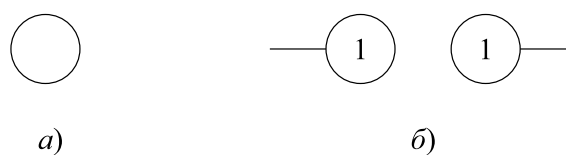


Рис. 6.20 – УГО символа соединителя

Соединители используются для избегания большого количества пересечений линий, когда управление должно передаться из одной удаленной части схемы в другую либо при разделении блок-схемы на несколько страниц. При этом соединители стоит сопровождать комментарием, куда именно передается управление.

Символ (см. рис. 6.21) отображает выход во внешнюю среду и вход из внешней среды (начало или конец схемы программы, внешнее использование и источник или пункт назначения данных).

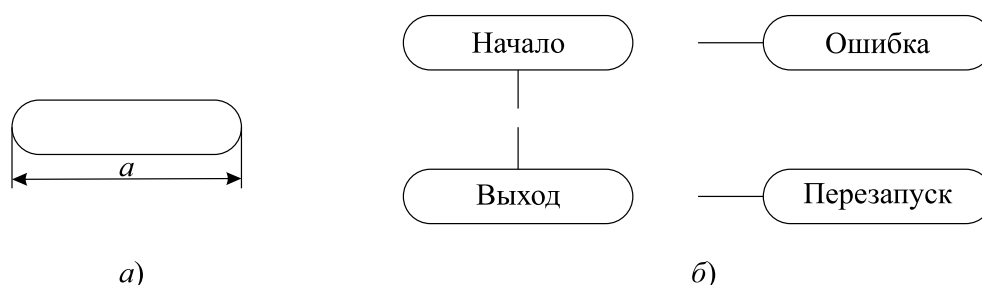


Рис. 6.21 – УГО символа терминатора

Одна блок-схема может содержать несколько входных или выходных терминаторов. При этом выходные терминаторы могут обозначать не только выход из алгоритма, но и такие специфические операции, как перезапуск алгоритма, выход с ошибкой, исключительная ситуация.

Символ (см. рис. 6.22) используют для добавления описательных комментариев или пояснительных записей в целях объяснения или примечаний. Пунктирные линии в символе комментария связаны с соответствующим символом или могут обходить группу символов. Текст комментариев или примечаний должен быть помещен около ограничивающей фигуры.

Символ (три точки) (см. рис. 6.23) используют в схемах для отображения пропуска символа или группы символов, в которых не определены ни тип, ни число символов. Символ используют только на линиях или между ними. Он применяется в схемах, представляющих общее решение какой-либо задачи с неизвестным числом повторений.

В схемах может использоваться идентификатор символов. Это некоторое значение, числовое или символьное, которое позволяет ссылаться на данный символ-процесс в других элементах документации. Идентификатор символа должен располагаться слева над символом (см. рис. 6.24).

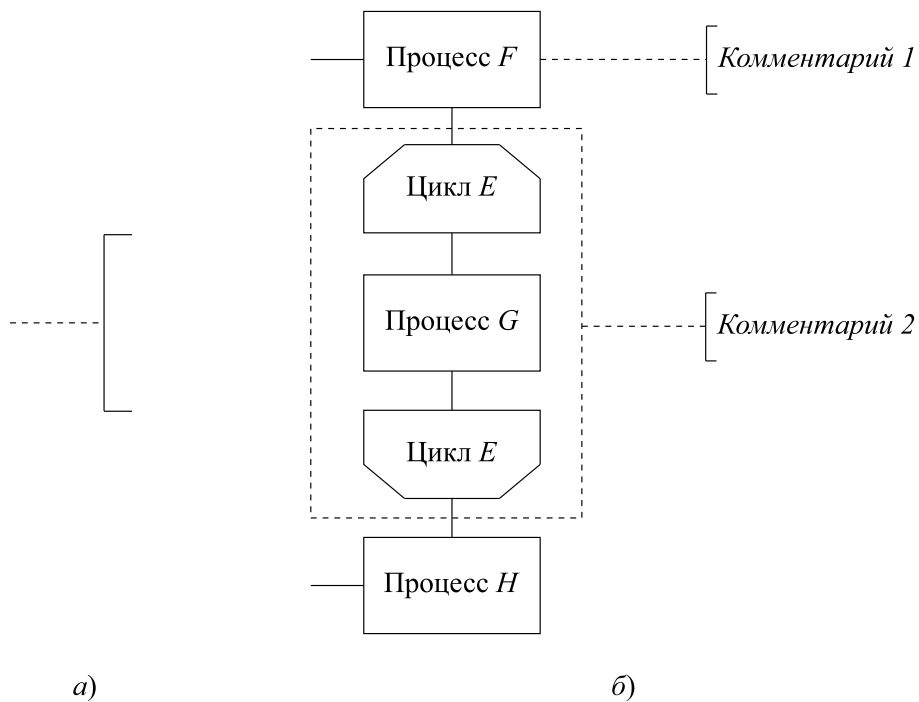


Рис. 6.22 – УГО символов: а) комментарий и б) пунктирных линий

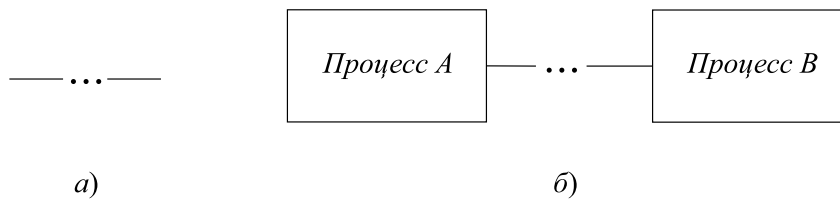


Рис. 6.23 – УГО символа пропуска

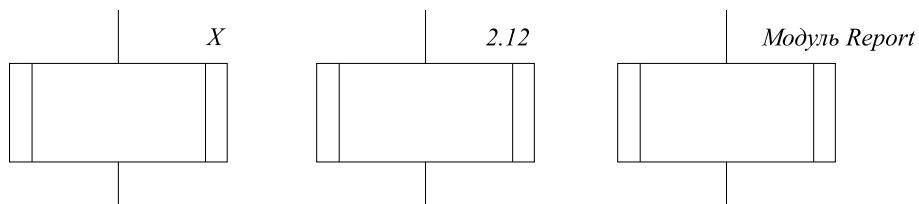


Рис. 6.24 – УГО символа идентификатора



## Выводы

Читатель может заметить, что ранее уже упоминалось, что излишняя документация может повредить проекту. С этим нельзя не согласиться. Главным критерием при использовании всех представленных видов описания программных систем являются два взаимоисключающих критерия, полнота описания и необходимая абстракция. С точки зрения полноты описания, при составлении документации нужно отметить все необходимые детали, которые в дальнейшем помогут разобраться в системе. С точки зрения необходимой абстракции, нужно скрыть ненужные де-

тали реализации и остановиться на том концептуальном уровне, который нужен для описания системы.

В заключение главы хотелось бы дать совет пользователю по использованию CASE (Computer Aided Software Engineering)-средств для построения представленных видов диаграмм:

- IDEF — для построения IDEF-диаграмм использовалась программа ERwin Process Modeler. Программа не совсем удобна, однако авторам не удалось найти более удобный редактор для IDEF-диаграмм.
- UML — для построения UML-диаграмм, с точки зрения авторов, более удобным является Enterprise Architect фирмы Sparks. Однако есть ещё несколько программных продуктов, которые можно использовать, т. к. они поддерживают последние нововведения текущего стандарта UML, но их поиск мы оставим читателю.
- Построение блок-схем — для построения корректных с точки зрения ГОСТа блок-схем авторами не было найдено ни одного специализированного ПО. Ближайшим продуктом для создания блок-схем можно назвать Microsoft Visio, содержащий готовые наборы элементов, приближенных к ГОСТу.



## Контрольные вопросы по главе 6

1. В чем разница обозначения циклов в блок-схемах и диаграммах деятельности?
2. Каково назначение терминалов в блок-схемах?
3. В каком порядке наиболее правильно будет составлять UML-диаграммы разрабатываемого продукта?
4. Для описания каких аспектов системы используется IDEF3?
5. Какую роль в диаграммах вариантов использования играют точки расширения?

---

## Глава 7

# ТЕХНИКИ НАПИСАНИЯ И ПОДДЕРЖКИ КОДА

---

В данной главе будет рассмотрен прикладной аспект разработки ПО, а именно — написание кода и проектирование архитектуры программы. Практически все участники команды, вовлеченной в разработку ПО, с ним сталкиваются. В этой главе будут рассмотрены паттерны проектирования ПО — шаблоны, позволяющие «сконструировать» архитектуру программы так, чтобы обеспечить ее гибкость и расширяемость. Также будут представлены «антипаттерны» — яркий пример того, как НЕ надо делать ПО, в основном, они состоят из типичных ошибок начинающих программистов. Так же будут даны некоторые рекомендации по качеству кода, будет рассмотрен рефакторинг и представлены рекомендации по оформлению кода. И конечно, будут рассмотрены инструменты, позволяющие в некоторой степени облегчить и даже автоматизировать написание правильного кода.

### 7.1 Паттерны проектирования

Разработка объектно-ориентированных программ — чрезвычайно сложное дело. Необходимо проанализировать систему, провести ее декомпозицию, представить систему в виде набора сущностей и определить, каким образом они взаимодействуют друг с другом. Впоследствии именно это мы будем подразумевать под архитектурой программы. Создание архитектуры не представляет сложности в простых проектах, где программа включает в себя 10–20 классов, однако для сложных проектов, в которых могут участвовать от 50 классов, определить их взаимодействие уже проблематично. Архитектура должна в первую очередь соответствовать решаемой задаче, при этом она должна быть достаточно «общей», чтобы впоследствии была возможность добавлять в программу новую функциональность, без изменений (или с минимальными изменениями) существующих модулей. При этом не стоит забывать, что основное преимущество ООП — повторное использо-

вание кода, то есть то, что программист написал для одного приложения, он может использовать и для другого, абсолютно не связанного с первым приложением. Соответственно, это тоже необходимо учитывать при проектировании архитектуры ПО.

Любой, даже самый опытный, программист скажет, что невозможно с первого раза создать достаточно гибкую и расширяемую архитектуру с возможностью повторного использования отдельных ее модулей. Именно поэтому процесс разработки ПО почти всегда итеративный. Однако чем опытней программист, тем меньше итераций ему необходимо для создания «удачной» архитектуры. Это обусловлено тем, что программист редко начинает разработку с нуля. И это обусловлено не только использованием существующих библиотек, но и тем, что со временем программист находит наиболее «удачные» подходы для обеспечения гибкого взаимодействия сущностей в программе. По сути это и называется паттерном проектирования.



.....  
**Паттерн проектирования** (шаблон проектирования) — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.  
.....

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код, это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конкретные классы или объекты приложения будут использоваться.

Канонической книгой с описанием всех паттернов с примерами является [51]. Для более глубокого изучения паттернов проектирования авторы данного учебного пособия советуют обратиться именно к ней. В данном пособии авторы приводят свою интерпретацию паттернов, с упрощенным описанием и примерами, адаптированными для .NET Framework.

К основным преимуществам шаблонов можно отнести:

- снижение сложности проектирования за счет использования готовых абстракций;
- облегчение коммуникации между разработчиками. Так как у каждого шаблона есть имя, то вместо описания архитектуры конкретного модуля, можно сослаться на конкретный паттерн, который будет использоваться при разработке данного модуля.

Однако знание паттернов несет и некоторую опасность:

- слепое следование некоторому выбранному шаблону может привести к усложнению программы;
- у разработчика может возникнуть желание попробовать некоторый шаблон в деле без особых оснований.

Стоит сказать о классификации паттернов проектирования. Существует ряд так называемых «основных» паттернов, которые могут быть применены при проектировании любой системы независимо от ее специализации, именно им будет уде-

лено наибольшее внимание. Однако существует множество других, более специализированных, паттернов, например паттерны параллельного программирования, паттерны web-приложений, enterprise паттерны (паттерны проектирования бизнес-приложений) и др.

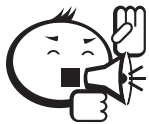
«Основные» паттерны в свою очередь делятся на три группы:

- порождающие паттерны предназначены для создания новых объектов в системе;
- структурные паттерны решают задачи компоновки системы на основе классов и объектов;
- паттерны поведения предназначены для распределения обязанностей между объектами в системе. *Из-за ограниченного объёма эти паттерны в данном учебном пособии не будут рассмотрены. Рекомендуется ознакомиться с ними самостоятельно.*

Рассмотрим каждую группу подробнее.

### 7.1.1 Порождающие паттерны

Порождающие паттерны абстрагируют процесс создания экземпляров конкретных объектов. Это позволяет не привязываться к ограниченному количеству взаимодействий сущностей, а описать некоторое количество базовых взаимодействий, с помощью композиции которых можно получать любое количество более сложных.



.....  
 Порождающие шаблоны позволяют скрывать реализацию и взаимодействие конкретных классов, существующих в системе. Все что система «знает» об этих классах — интерфейс. Таким образом, порождающие шаблоны обеспечивают большую гибкость при решении вопроса о том, что создается, кем создается, как и когда. Это позволяет собирать системы с различной функциональностью как на этапе компиляции, так и на этапе выполнения.  
 .....

#### **Абстрактная фабрика.**

*Абстрактная фабрика* — порождающий паттерн проектирования, предоставляющий интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

UML-диаграмма паттерна *Абстрактная фабрика* представлена на рисунке 7.1.

Данный паттерн применяется, если:

- система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты;
- входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения;
- система должна конфигурироваться одним из семейств составляющих ее объектов;



- требуется предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию, что позволяет избежать привязки клиентского кода к конкретной технологии.

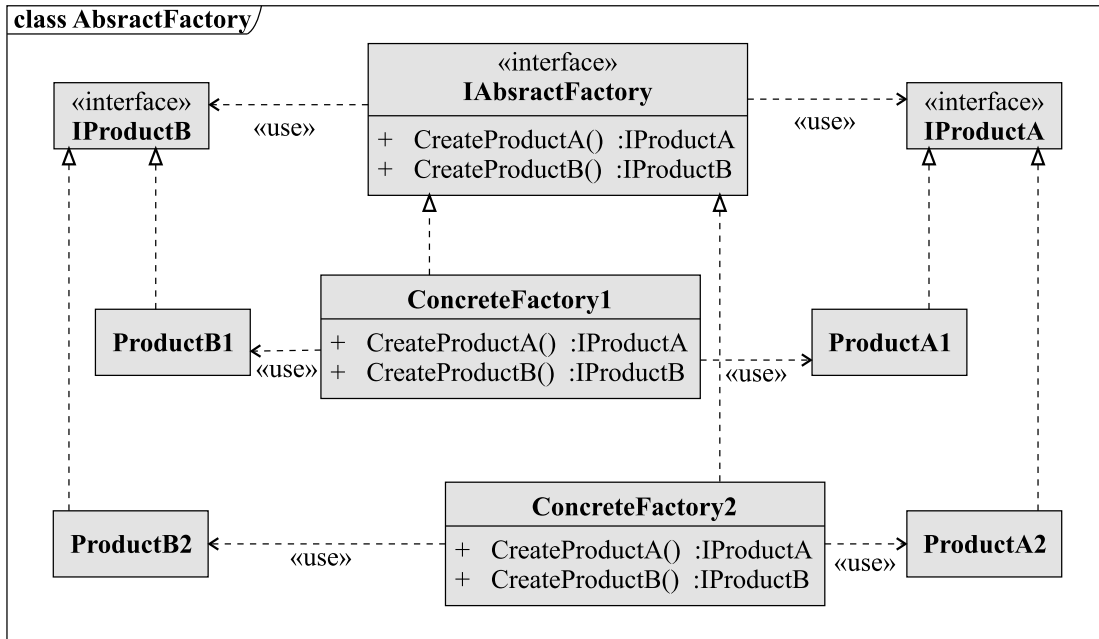


Рис. 7.1 – UML-диаграмма паттерна *Абстрактная фабрика*

Рассмотрим преимущества и недостатки данного паттерна. Использование абстрактной фабрики позволяет легко заменить одно семейство продуктов другим, при этом гарантируя их сочетаемость. Так как все созданные продукты системы зависят от экземпляра конкретной фабрики, то можно легко изменить типы продуктов, используя экземпляр другой абстрактной фабрики. У данного паттерна имеется существенный недостаток — сложность поддержки новых типов продуктов. Интерфейс *Абстрактной фабрики* фиксирует в себе все возможные для создания продукты, и если впоследствии понадобится добавить еще типов, необходимо будет изменять интерфейс фабрики и реализацию всех существующих ее подклассов.



## Пример

Допустим, мы реализуем приложение, в котором необходимо обеспечить поддержку различных стилей оформления. Стиль включает в себя внешний вид элементов пользовательского интерфейса, например кнопок, полей ввода, полос прокрутки и так далее.

При решении этой задачи с помощью паттерна *Абстрактная фабрика* мы получим следующую архитектуру (рис. 7.2).

Метод `SetStyle` класса `MainWindow` выглядит следующим образом:

```
private void SetStyle(IStyleFactory factory)
```

```

{
    _toolbar = factory.GetToolBar();
    _button = factory.GetButton();
    /*
    Инициализация других зависимых компонентов
    */
}

```

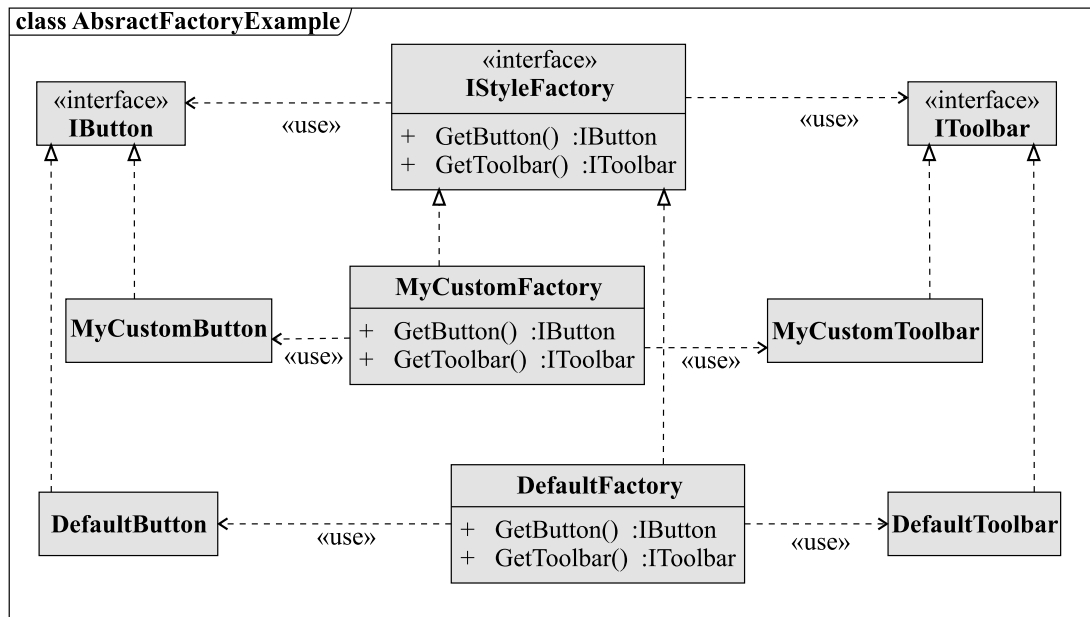


Рис. 7.2 – UML-диаграмма модуля настройки стилей

При таком подходе мы можем полностью изменить визуальную часть программы следующим вызовом:

```
SetStyle(new MyCustomStyle());
```

### Строитель.

*Строитель* (builder) – порождающий шаблон проектирования, позволяющий отделить конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.

UML-диаграмма паттерна *Строитель* представлена на рисунке 7.3.

Паттерн *Строитель* отделяет алгоритм пошагового создания сложного объекта от его внешнего представления так, что с помощью одного и того же алгоритма можно получать разные представления этого продукта.

Для этого паттерн *Строитель* определяет алгоритм поэтапного создания продукта в специальном классе *Director* (распорядитель), а ответственность за координацию процесса сборки отдельных частей продукта возлагает на иерархию классов *Builder*. В этой иерархии базовый класс *Builder* объявляет интерфейсы для построения отдельных частей продукта, а соответствующие подклассы *ConcreteBuilder* их реализуют подходящим образом, например создают или получают нужные ресур-

сы, сохраняют промежуточные результаты, контролируют результаты выполнения операций.

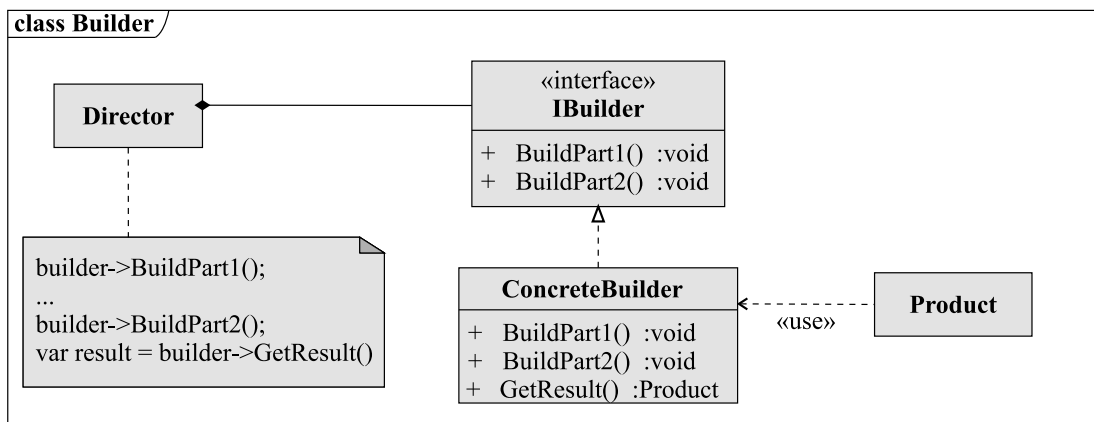
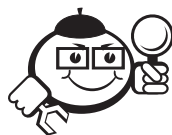


Рис. 7.3 – UML-диаграмма паттерна *Строитель*

Преимущества данного паттерна:

- позволяет изменять внутреннее представление продукта. Поскольку объект конструируется через абстрактный интерфейс, то для его изменения достаточно определить еще один класс строителя;
- изолирует код, реализующий конструирование и представление. Использование данного паттерна позволяет скрыть способ конструирования и внутреннюю структуру объекта, так как клиенту ничего не надо знать о классах, определяющих внутреннюю структуру объекта, они даже отсутствуют в интерфейсе строителя;
- дает более тонкий контроль над процессом конструирования. В отличие от других порождающих паттернов, конструирующих объект целиком, здесь мы имеем возможность конструировать объект шаг за шагом.

В целом данный паттерн похож на *Абстрактную фабрику*, поэтому необходимо указать их различия — если абстрактная фабрика концентрируется на построении СЕМЕЙСТВА родственных объектов, то *Строитель* позволяет проводить пошаговое построение ОДНОГО объекта.



## Пример

Допустим, надо определить процесс построения различных видов домов. Дома отличаются количеством комнат и этажей.

На рисунке 7.4 представлена UML-диаграмма архитектуры программы, решающей данную задачу.

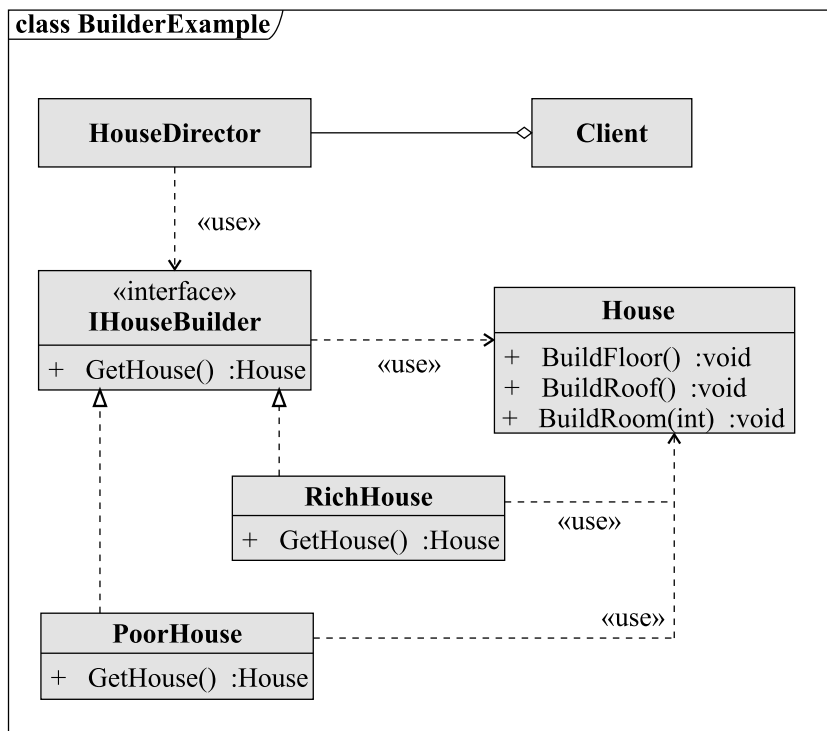


Рис. 7.4 – UML-диаграмма архитектуры программы, позволяющей строить различные виды домов

Далее представлена реализация классов данной архитектуры.

```

interface IHouseBuilder
{
    House GetHouse();
}
class PoorHouse : IHouseBuilder
{
    public House GetHouse()
    {
        var h = new House();
        h.BuildFloor();
        h.BuildRoom(1);
        h.BuildRoom(1);
        h.BuildRoof();
        return h;
    }
}
class RichHouse : IHouseBuilder
{
    public House GetHouse()
    {
        var h = new House();
        h.BuildFloor();
    }
}
  
```

```
        h.BuildRoom(1);
        h.BuildRoom(1);
    h.BuildFloor();
        h.BuildRoom(2);
        h.BuildRoom(2);
    h.BuildFloor();
        h.BuildRoom(3);
        h.BuildRoom(3);
        h.BuildRoof();
        return h;
    }
}
class HouseDirector
{
    public House BuildHouse(IHouseBuilder builder)
    {
        return builder.BuildHouse();
    }
}
class Client
{
    private void DoSomething()
    {
        var director = new HouseDirector();
        var richHouse = director.BuildHouse(new RichHouse());
        var poorHouse = director.BuildHouse(new PoorHouse());
    }
}
```

Проанализируем данный фрагмент кода. У нас существует абстракция дома (класс *House*). Также определены 2 строителя, позволяющие строить разные дома — один одноэтажный с двумя комнатами (*PoorHouse*), второй — трехэтажный с тремя комнатами на каждом этаже. При необходимости создания дома с другими параметрами необходимо только создать новый класс-строитель и определить в нем сборку на основе доступных методов класса *House*.

#### **Фабричный метод.**

*Фабричный метод* (Factory method, также известен как Virtual constructor) — порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Данный шаблон используется в следующих случаях:

- класс заранее не знает, какие объекты необходимо будет создавать, т.к. возможны варианты реализации;
- класс спроектирован так, что спецификация порождаемого объекта определяется только в наследниках;

- класс выделяет и делегирует часть своих функций вспомогательному классу. При этом необходимо скрыть его реализацию для достижения большей гибкости или возможности расширения функциональности.

На рисунке 7.5 представлена UML-диаграмма данного паттерна.

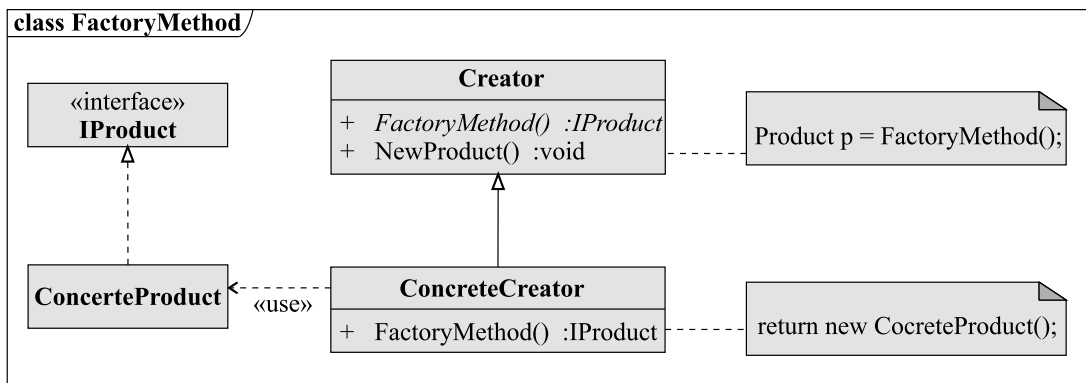


Рис. 7.5 – UML-диаграмма паттерна *Фабричный метод*



## Пример

Рассмотрим простой менеджер документов, который позволяет работать с текстовыми файлами. В функции менеджера входит сохранение и загрузка файла. Стоит отметить, что любой текстовый файл может сохраняться различными способами, например простое сохранение и сохранение с дополнительным кодированием. При этом необходима возможность работать с различными форматами документов.

Ниже представлена архитектура, иллюстрирующая данную задачу (рис. 7.6).

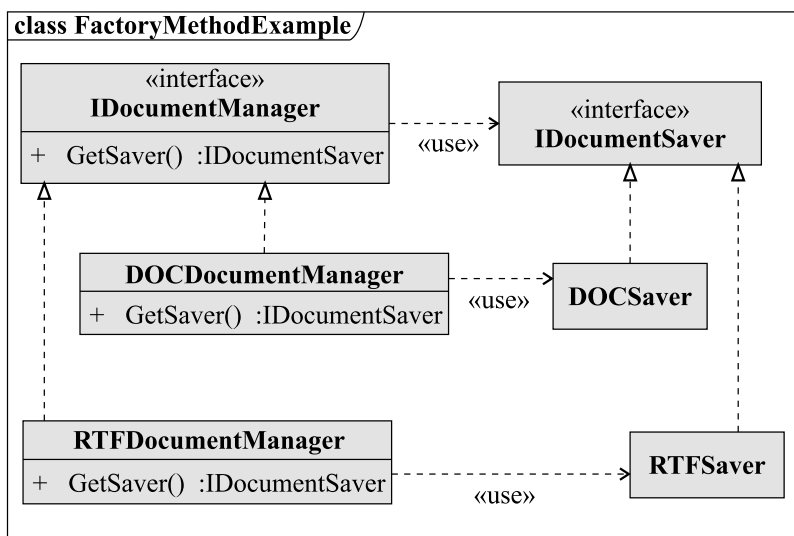


Рис. 7.6 – UML-диаграмма модуля «менеджер документов»

В системе определен интерфейс для менеджера документов *IDocumentManager* и два его потомка *RTFDocumentManager* и *DOCDocumentManager*, позволяющие работать с документами формата .rtf и .doc соответственно. Также определен интерфейс для класса, отвечающего за сохранение конкретного типа документа. Рассмотрим код реализации классов *RTFDocumentManager* и *DOCDocumentManager*.

```
class RTFDocumentManager:IDocumentManager
{
public IDocumentSaver GetSaver()
{
    return new RTFSaver();
}
}
class DOCDocumentManager:IDocumentManager
{
public IDocumentSaver GetSaver()
{
    return new DOCSaver();
}
}
```

Функция создания и сохранения документа представлена ниже.

```
private void DoSomething(string filename)
{
    IDocumentManager manager = new RTFDocumentManager();
    IDocumentSaver saver = manager.GetSaver();
    saver.Save(filename);
}
```

В данном примере создание менеджера и сохранение идут вместе, поэтому можно было бы создать экземпляр класса *RTFSaver* явно, однако в следующем примере явное создание становится невозможным.

```
private void DoSomething(IDocumentManager manager,
string filename)
{
    IDocumentSaver saver = manager.GetSaver();
    saver.Save(filename);
}
```

В данном случае тип менеджера неизвестен, поэтому нельзя создавать экземпляр класса для сохранения явно, так как он может быть не совместим с экземпляром текущего менеджера.

В предыдущем примере использовались различные классы для представления каждого формата, однако возможен вариант, когда у нас есть одно представление и его надо сохранять различными способами. В этом случае могут использоваться параметризованные фабричные методы.

```
class DocumentManager
{
    public enum DocumentFormat {DOC, RTF};
```

```
public IDocumentSaver GetSaver(DocumentFormat format)
{
    switch (format)
    {
        case DocumentFormat.DOC: return new DOCSaver();
        break;
        case DocumentFormat.RTF: return new RTFSaver();
        break;
    }
}
```

Стоит отметить, что в языках с поддержкой шаблонов существует еще один вариант реализации фабричного метода.

```
class DocumentManager
{
    public enum DocumentFormat {DOC, RTF};
    public IDocumentSaver GetSaver<T>()
    where T : IDocumentSaver
    {
        return new T();
    }
}
```

В этом случае метод сохранения будет реализован следующим образом.

```
private void DoSomething(DocumentManager manager,
string filename)
{
    IDocumentSaver saver = manager.GetSaver<RTFSaver>();
    saver.Save(filename);
}
```

### Прототип.

*Прототип* (Prototype) — порождающий шаблон проектирования, который позволяет задавать виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа. Данный подход может быть выгодным в том случае, если создавать объект через конструктор затратно для приложения (так как в этом случае будут вызваны конструкторы всей иерархии предков).

Паттерн применяется в следующих случаях:

- инстанцируемые классы определяются во время выполнения, например с помощью динамической загрузки, в этом случае у клиента нет возможности создать объект через его конструктор;
- для того чтобы избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов (например, в случае использования фабричного метода на каждый продукт придется создавать класс, который его порождает);
- экземпляры класса могут находиться в одном из нескольких различных состояний. Может оказаться удобнее установить соответствующее число про-



тотипов и клонировать их, а не инстанцировать каждый раз класс вручную в подходящем состоянии.

У данного паттерна существует недостаток, что в некоторых случаях достаточно сложно реализовать оператор копирования, так как объект может агрегировать достаточно сложные структуры.

На рисунке 7.7 представлена UML-диаграмма данного паттерна.

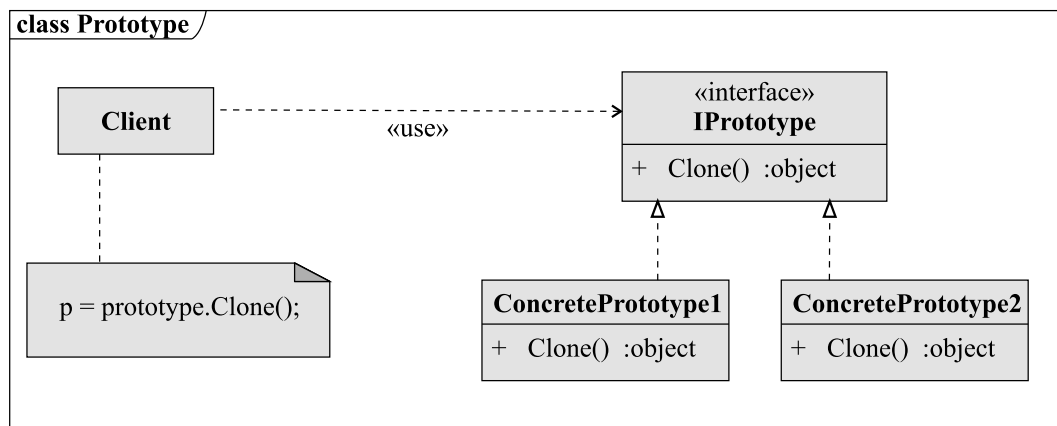


Рис. 7.7 – UML-диаграмма паттерна *Прототип*



## Пример

Необходимо разработать систему, которая работает с результатами измерений некоторого прибора. Результаты хранятся в форматированном текстовом файле. Система должна считывать файл и представлять его содержимое в удобном для приложения виде. Дело в том, что для таких задач применяются различные синтаксические анализаторы, которые являются достаточно ресурсоемкими, и часто процесс анализа занимает значительное время. Поэтому слишком затратным является открытие одного и того же файла 2 раза для получения 2-х объектов с одинаковыми измерениями. Именно в этом случае применение паттерна *Прототип* является обоснованным, так как позволит сэкономить ресурсы ПК и ускорит работу программы.

UML-диаграмма классов примера представлена на рисунке 7.8.

В данном случае код клиента будет выглядеть следующим образом.

```

public void DoSomething()
{
    var measurement1 =
    new Measurement("ALotOfDifferentMeasurements.txt");
    var measurement2 = measurement1.Clone();
}
  
```

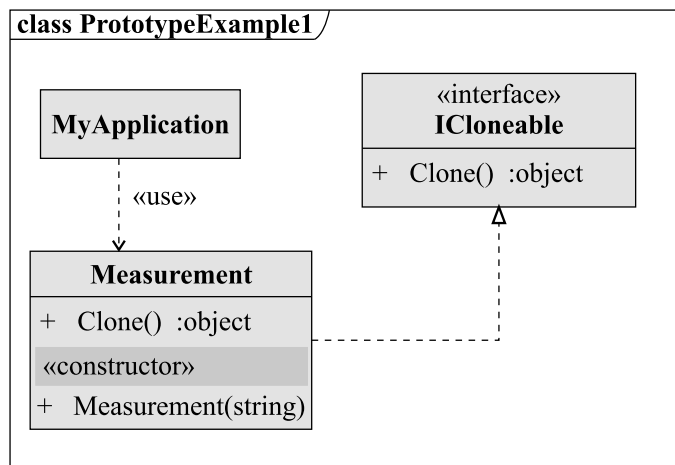


Рис. 7.8 – UML-диаграмма модуля работы с измерениями



## Пример

Необходимо спроектировать векторный графический редактор с функцией выделения набора фигур и их копированием.

На рисунке 7.9 представлена UML-диаграмма архитектуры данного редактора.

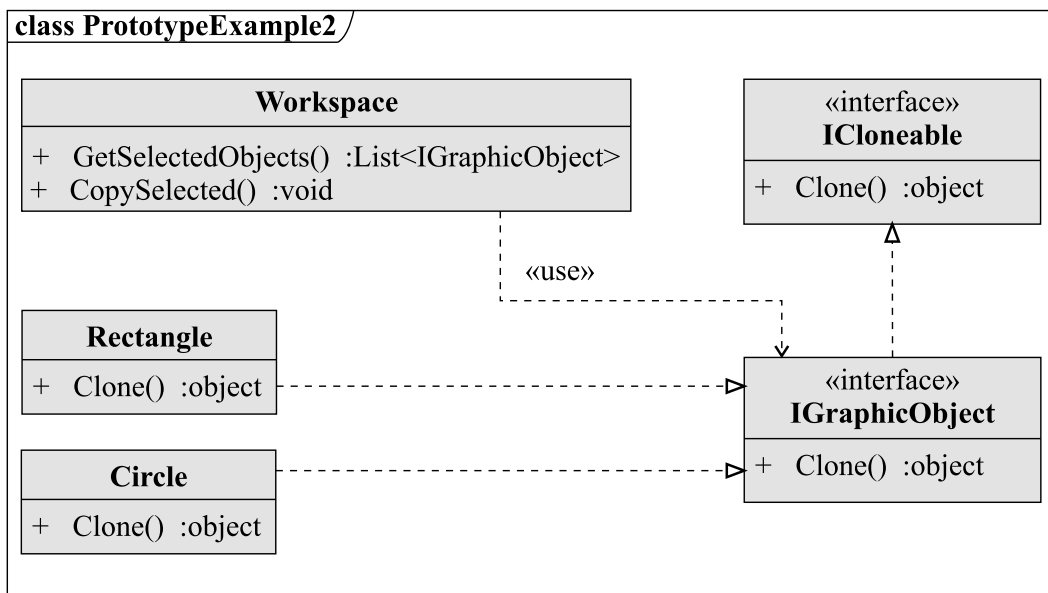


Рис. 7.9 – UML-диаграмма архитектуры векторного графического редактора

Основным классом является Workspace, на котором могут быть расположены различные объекты, которые хранятся в виде интерфейсов IGraphicObject. Сложность копирования в том, что выделены могут быть разнообразные объекты и система не «знает», какие именно, поэтому в данном случае невозможно воспользо-

ваться конструктором конкретных классов. Функция копирования будет реализована следующим образом.

```
public void CopySelected()
{
    List<IGraphicObject> selectedObjects =
        GetSelectedObjects();
    List<IGraphicObject> newObjects =
        new List<IGraphicObject>();
    foreach(var obj in selectedObjects)
    {
        var temp = obj.Clone();
        //Различные манипуляции с новым объектом,
        //например изменение определенных параметров
        newObjects.Add(temp);
    }
}
```

### **Одиночка.**

*Одиночка* (Singleton) — порождающий шаблон проектирования, гарантирующий, что в приложении будет единственный экземпляр класса с глобальной точкой доступа. Данный шаблон применяется, когда в системе должен существовать только один класс и все части системы должны иметь к нему доступ. По сути этот паттерн является дальнейшим развитием глобальных переменных и статических классов.

*Одиночка* обладает следующим главным преимуществом по сравнению с глобальными переменными — он не засоряет пространство имен, то есть все, к чему необходим глобальный доступ, инкапсулируется в экземпляре объекта.

По сравнению со статическим классом (класс, в котором все поля и методы статические) главное преимущество в том, что работа идет с ЭКЗЕМПЛЯРОМ объекта, следовательно:

- от класса-одиночки возможно наследоваться;
- его можно передавать в качестве параметра;
- существует возможность контролировать время жизни объекта (ленивая инициализация, см. далее);
- возможна сериализация (для высокоуровневых языков наподобие C# и Java).

На рисунке 7.10 представлена UML-диаграмма данного паттерна.

Типичная реализация представлена ниже.

```
public class Singleton
{
    private static readonly Singleton _instance =
        new Singleton();
    public static Singleton Instance
    {
        get {return _instance;}
    }
}
// Защищенный конструктор нужен, чтобы предотвратить
```

```
// создание экземпляра класса Singleton
protected Singleton() {}
// Любые методы, к которым необходим глобальный доступ
public void DoSomething(){}
}
```

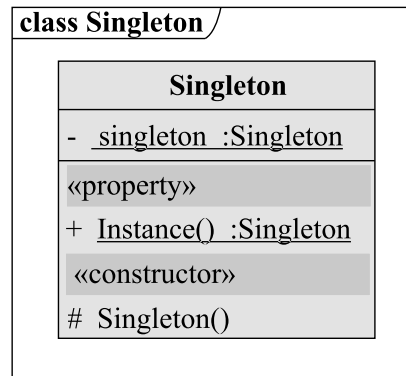


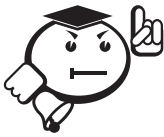
Рис. 7.10 – UML-диаграмма паттерна *Одиночка*

При такой реализации объект класса одиночки будет создаваться в момент инициализации статических полей — инициализация приложения. Существует реализация, при которой экземпляр будет создан, при первом обращении к классу-одиночке такой подход называется «ленивая инициализация» (lazy initialization).

```
namespace Singleton
{
    public class Singleton
    {
        private static Singleton _instance;
        public static Singleton Instance
        {
            get
            {
                if (_instance == null)
                {
                    _instance = new Singleton();
                }
                return _instance;
            }
        }
        protected Singleton() {}
        // Любые методы, к которым необходим глобальный доступ
        public void DoSomething(){}
    }
}
```

Независимо от реализации, к объекту-одиночке можно обратиться из любой части программы следующим вызовом:

```
Singleton.Instance.DoSomething();
```



.....  
 Существуют несколько вариаций данного паттерна, например шаблонная реализация, потокобезопасная реализация, но в данном по-  
 собии они рассматриваться не будут.  
 .....

### 7.1.2 Структурные паттерны

В структурных паттернах рассматривается вопрос, как из классов и объектов формируются крупные структуры. Рассматривают два вида структурных паттернов:

- уровня класса. Использует наследование для составления композиций из интерфейсов и реализаций. Как пример, множественное наследование, в результате получается потомок, обладающий всеми свойствами родителей. Структуры на основе наследования получают статичными;
- уровня объекта. Использует композицию, компоуя объекты для получения новой функциональности. Композиция позволяет получать структуры, которые можно изменять во время выполнения.

#### Адаптер.

*Адаптер* (Adapter, также известен как Wrapper) — структурный паттерн проектирования, преобразует интерфейс одного класса в интерфейс другого, который использует клиент. Позволяет обеспечить совместимую работу классов с несовместимыми интерфейсами. Особенно полезен при встраивании сторонних библиотек в приложение, так как интерфейс классов сторонних библиотек не всегда позволяет эффективно их использовать в рамках пользовательского приложения.

Преимуществом паттерна *Адаптер* является то, что он позволяет повторно использовать уже имеющийся код, адаптируя его несовместимый интерфейс к виду, пригодному для использования. Недостатком является то, что задача преобразования интерфейсов может оказаться непростой в случае, если клиентские вызовы и (или) передаваемые параметры не имеют функционального соответствия в адаптируемом объекте.

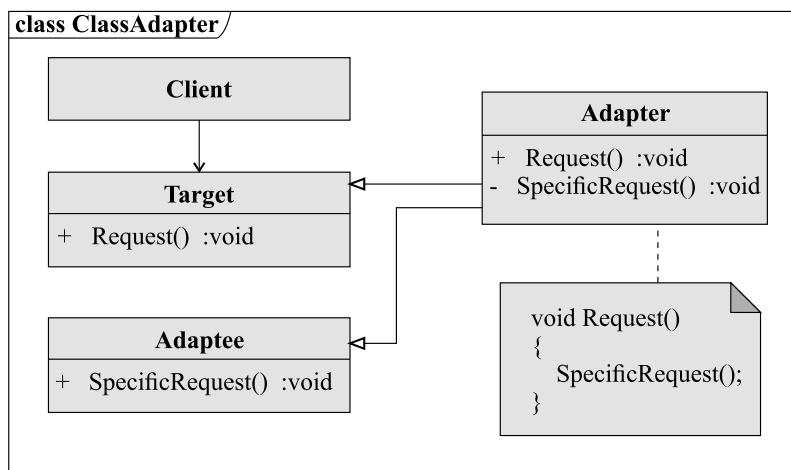
На рисунке 7.11 представлена UML-диаграмма данного паттерна.



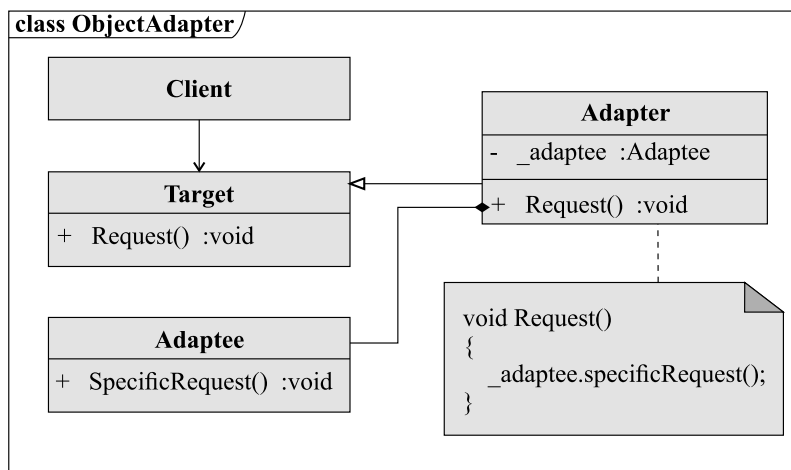
#### Пример

.....  
 Необходимо разработать приложение, в котором будет ряд пользовательских элементов управления, например кнопки поля ввода. Если с реализацией кнопки нет проблем, а поле ввода по каким-то причинам необходимо использовать из сторонней библиотеки, причем в приложении размер элемента управления определяется двумя точками (верхний левый угол и правый нижний), то в библиотеке используется задание размеров через начальную точку, длину и ширину элемента управления.  
 .....

Архитектура решения представлена на рисунке 7.12.



a)



б)

Рис. 7.11 – UML-диаграмма паттерна *Адаптер*: а) уровня класса; б) уровня объекта

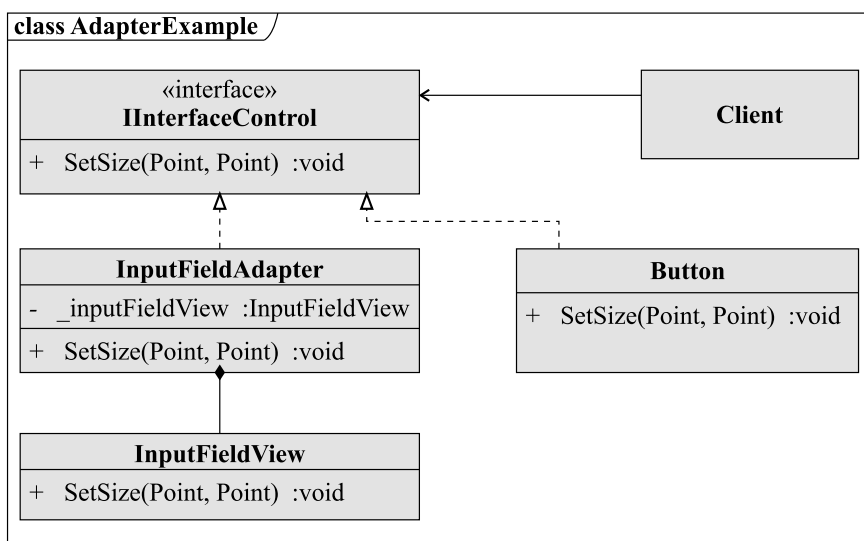


Рис. 7.12 – UML-диаграмма архитектуры тестовой программы

Ниже представлена реализация класса *InputFieldAdapter*.

```
class InputFieldAdapter : IInterfaceControl
{
    private InputFieldView _inputFieldView =
        new InputFieldView();
    public void SetSize(Point leftHighCorner,
        Point rightLowCorner)
    {
        var width = rightLowCorner.X - leftHighCorner.X;
        var height = rightLowCorner.Y - leftHighCorner.Y;
        inputFieldView.SetSize(leftHighCorner, width, height);
    }
}
```

### Мост.

*Мост* (Bridge) — структурный шаблон проектирования, позволяющий разделять абстракцию и реализацию так, чтобы они могли изменяться независимо. Когда абстракция и реализация разделены, они могут изменяться независимо. Другими словами, при реализации через паттерн *Мост* изменение структуры интерфейса не мешает изменению структуры реализации (рис. 7.13).

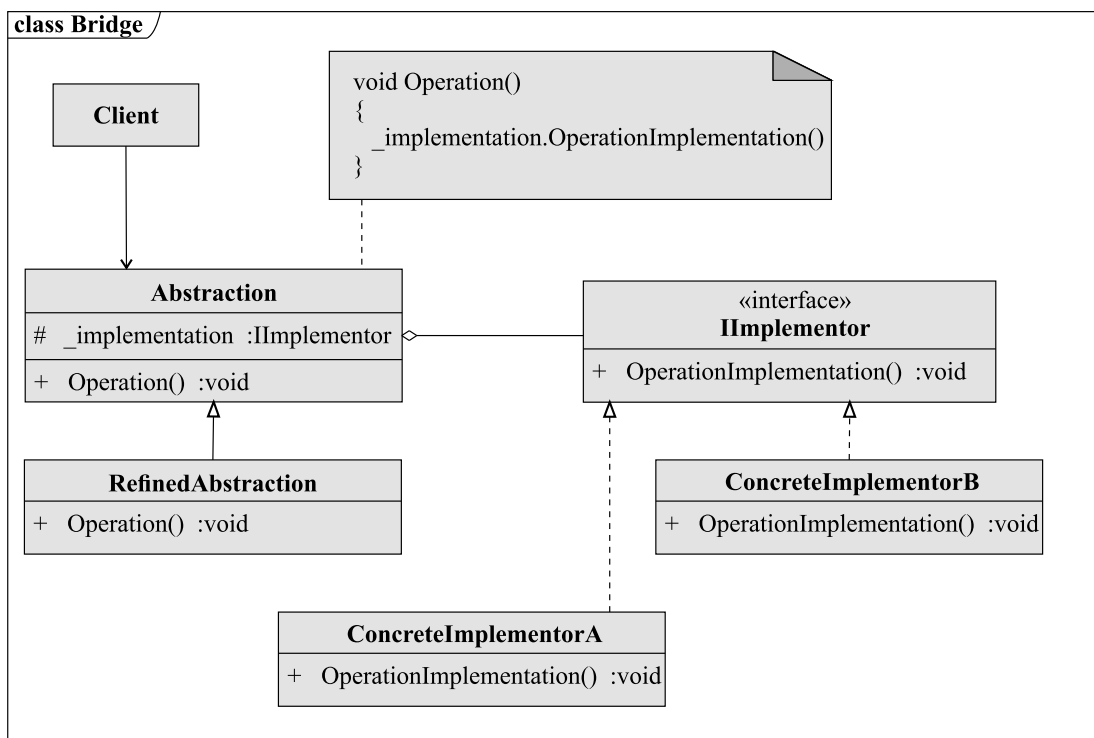


Рис. 7.13 – UML-диаграмма паттерна *Мост*



### Пример

Необходимо разработать оконное приложение, которое может работать на основе различных оконных систем (например, в случае использования различных

ОС, таких как Windows и MacOS). Очевидно, что в этом случае достаточно создать два потомка класса Window для каждой ОС. Однако впоследствии необходимо добавить еще 2 вида окна, например Dialog и Widget. Очевидно, что они тоже являются наследниками класса Window. Скорее всего, понадобится сделать реализацию таким образом, чтобы каждый из этих подклассов работал на обеих ОС. Получается, что необходимо создать еще несколько потомков, таких как *WidgetMacOS*, *WidgetWindows*, *DialogMacOs* и *DialogWindows*. Это приведет к разрастанию иерархии наследования и затруднит дальнейшее расширение программы. Для решения этой проблемы используется паттерн *Мост*. В этом случае абстракции окна выделены в отдельную иерархию наследования, а реализации в отдельную, что позволяет легко манипулировать абстракциями и расширять реализации.

Пример конечной архитектуры приведен на рисунке 7.14.

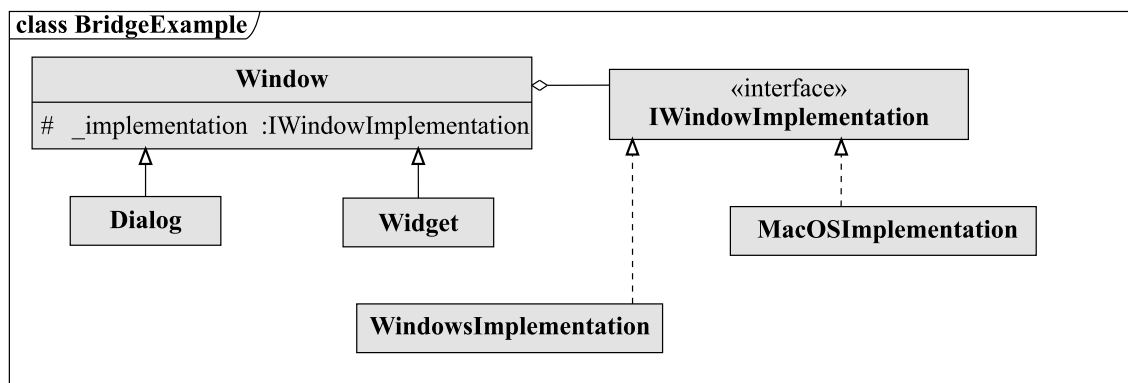


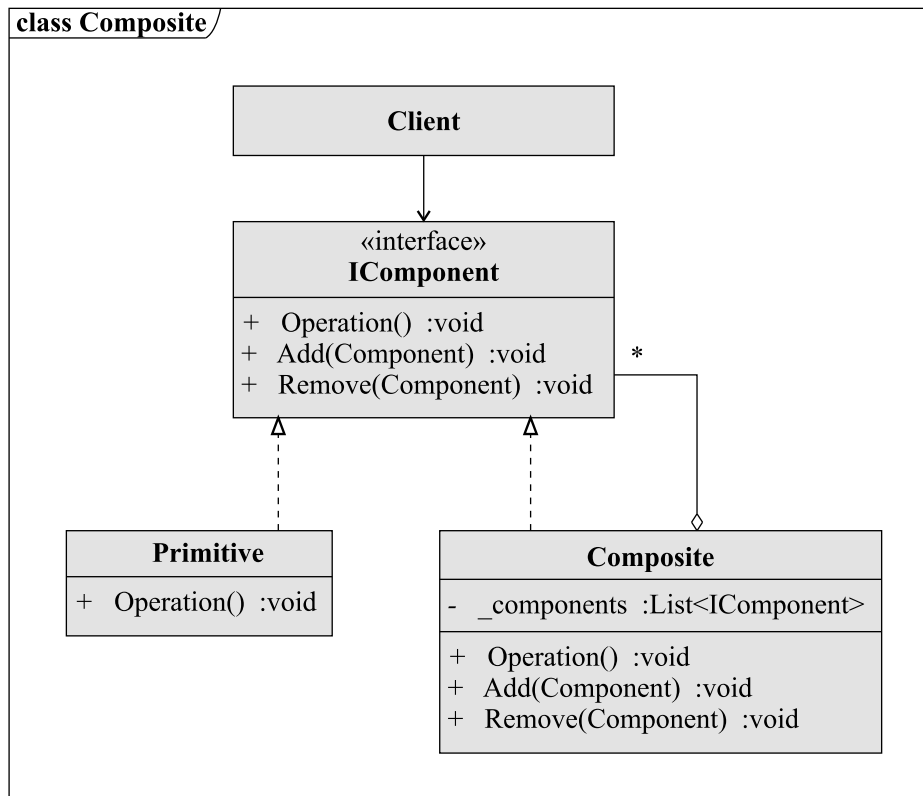
Рис. 7.14 – UML-диаграмма архитектуры кроссплатформенного оконного приложения с применением паттерна *Мост*

### Компоновщик.

*Компоновщик (Composite)* – структурный шаблон проектирования, относится к структурным паттернам, объединяет объекты в древовидную структуру для представления иерархии от частного к целому. *Компоновщик* позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково. Данный паттерн позволяет определить иерархию классов, которые могут состоять одновременно из примитивных и сложных объектов, упрощает архитектуру клиента, делает процесс добавления новых видов объекта более простым (рис. 7.15).

В данном случае класс *Component* представляет собой интерфейс, который представляет и примитивы, и контейнеры. В нем определены общие методы для всех элементов системы (для графических объектов это может быть метод Draw(), для систем моделирования сложных объектов – Simulate()) и методы для составных объектов, например операции для работы с потомками. Класс *Primitive* определяет собой далее неделимую сущность в системе, так как у него нет потомков, то в нем не реализуются методы работы с ними. Класс *Composite* агрегирует в себе объекты класса *Component*. Поскольку интерфейс *Composite* соответствует интерфейсу *Component*, то *Composite* может одинаково включать в себя как объекты *Primitive*, так и объекты *Composite*.



Рис. 7.15 – UML-диаграмма паттерна *Компоновщик*

Основной сложностью при использовании данного паттерна в проектировании является выделение интерфейса класса *Component*, так как он должен включать в себя как методы работы с составными объектами, так и неделимыми частями, соответственно, необходимо выделить как можно больше общих методов. Однако это вступает в противоречие с принципом проектирования классов — класс должен определять только логичные для него методы, а класс *Component* будет поддерживать множество операций, несвойственных для класса *Primitive*.

Данная проблема решается в зависимости от того, что важнее в данной системе:

- Прозрачность. Если определить методы работы с потомками в классе *Component*, то система становится прозрачной — все элементы системы можно трактовать однообразно. Однако в этом случае появляется опасность выполнения множества бессмысленных операций для объектов *Primitive*.
- Безопасность. В данном случае методы работы с потомками помещаются в *Composite*. Тем самым обеспечивая корректную работу в статически типизированных языках — попытка работать с потомками классов *Primitive* приведёт к ошибке компиляции.

Для паттерна *Компоновщик* важнее первое, но окончательное решение все же принимает проектировщик системы.



## Пример

Данный паттерн может найти применение в системе моделирования электрических цепей, потому что цепь может состоять как из конечных компонентов (резисторы, конденсаторы, диоды), так и из ранее собранных подцепей.

Архитектура представлена на рисунке 7.16. В данном случае мы не учитываем топологические особенности схемы (способы включения элементов).

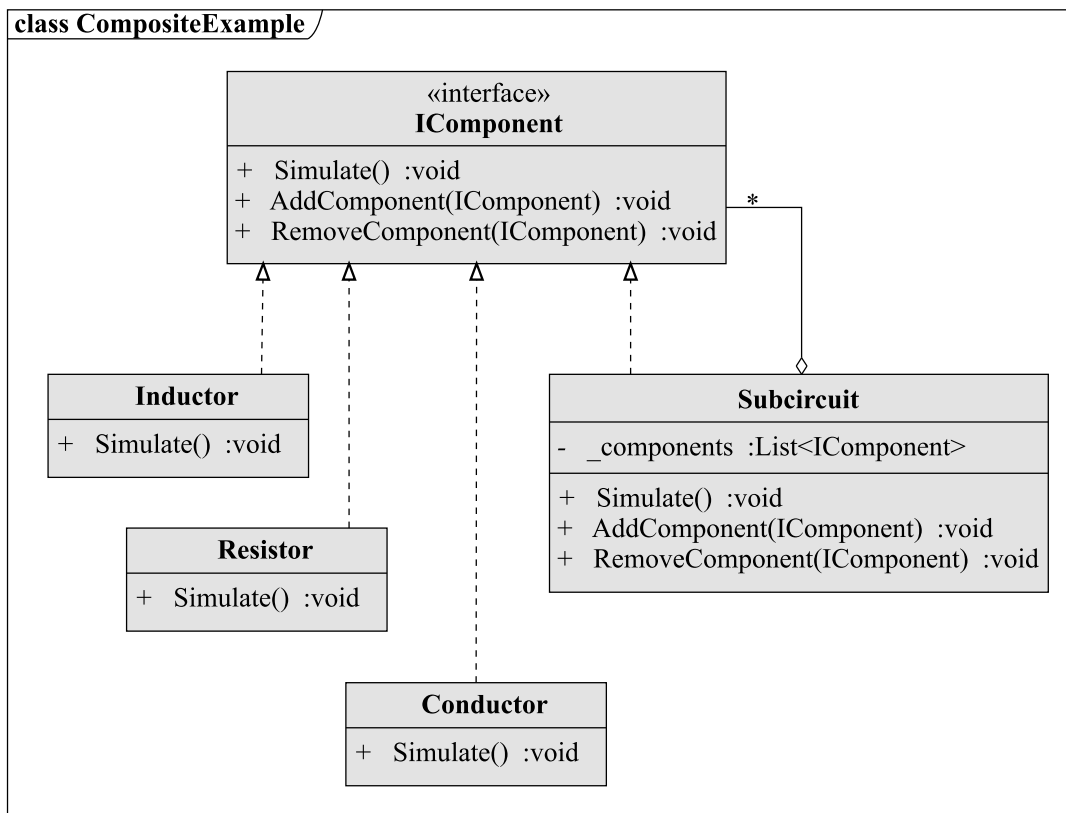


Рис. 7.16 – UML-диаграмма системы моделирования электрических цепей

Реализация класса *Subcircuit*:

```

public class Subcircuit: IComponent
{
    private list<IComponent> _components =
        new list<IComponent>();
    public SimulationResult Simulate()
    {
        SimulationResult result = new SimulationResult();
        Foreach (var component in _components)
            result.Compose(component.Simulate());
    }
}
  
```

```
public void AddComponent(IComponent component)
{
    _components.Add(component);
}
public void RemoveComponent(IComponent component)
{
    _components.Remove(component);
}
}
```

Пример построения и моделирования цепи представлен ниже.

```
public void CreateAndSimulateCircuit()
{
    var subcircuit1 = new Subcircuit();
    subcircuit1.Add(new Inductor());
    subcircuit1.Add(new Conductor());

    var subcircuit2 = new Subcircuit();
    subcircuit2.Add(new Resistor());
    subcircuit2.Add(subcircuit1);
    subcircuit2.Add(new Diod());

    var circuit = new Subcircuit();
    circuit.Add(new Conductor ());
    circuit.Add(subcircuit1);
    circuit.Add(new Resistor());
    circuit.Add(subcircuit2);
    circuit.Add(new Conductor ());

    var result = circuit.Simulate();
}
```

### Декоратор.

*Декоратор* (Decorator, также известен как Wrapper) — структурный шаблон проектирования, позволяющий добавлять объекту новые обязанности. Является альтернативой порождению подклассов с целью расширения функциональности.

Паттерн *Декоратор* используется:

- для динамического и прозрачного для клиентов добавления новых возможностей;
- для реализации возможностей, которые нужны не всем объектам и не всегда, для того, чтобы потом можно было легко их исключить;
- когда расширение путем порождения подклассов по каким-то причинам неудобно или невозможно. Иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведет к комбинаторному росту их числа (рис. 7.17). В других случаях определение класса может быть скрыто или почему-либо еще недоступно, так что породить от него подкласс нельзя.

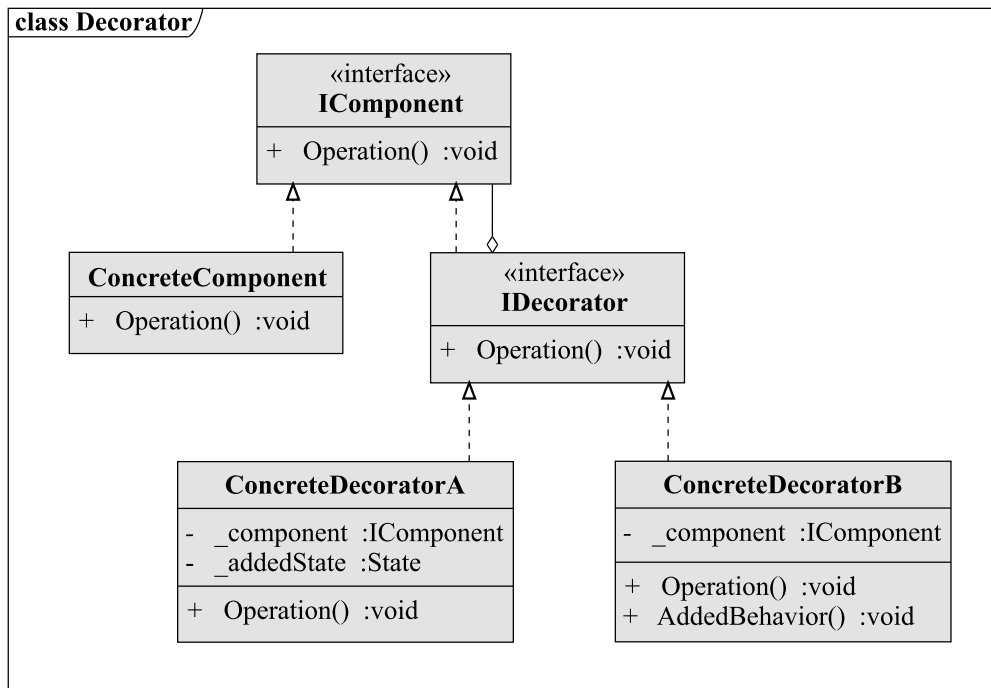


Рис. 7.17 – UML-диаграмма паттерна Декоратор

Данный паттерн обладает следующими достоинствами:

- Большая гибкость, чем у механизма наследования. Паттерн позволяет добавлять и удалять обязанности объекта на этапе выполнения программы.
- Позволяет избежать перегруженных функциями классов. Паттерн позволяет добавлять обязанности по мере необходимости, чем позволяет избежать создания классов, в которых реализованы все мыслимые комбинации свойств и поведений объектов.

Также можно выделить некоторые недостатки паттерна:

- декоратор и конечный объект не идентичны. Это вытекает из философии наследования — все порожденные потомки являются «равноправными», то есть при наследовании от класса *Animal* (животное) все его потомки будут являться животными, в данном же случае конечный объект и декоратор — различные сущности;
- порождает множество различных мелких объектов.



## Пример

Необходимо спроектировать текстовый редактор пользовательского интерфейса. В системе существуют элементы управления — кнопка, поле ввода. Затем в систему понадобилось добавить возможность расширения существующих компонентов, например поле ввода с полосой прокрутки, поле ввода с рамкой, кнопка с рамкой, и т. д. Можно решить данную задачу с помощью наследования.

В данном случае получится иерархия наследования, представленная на рисунке 7.18.

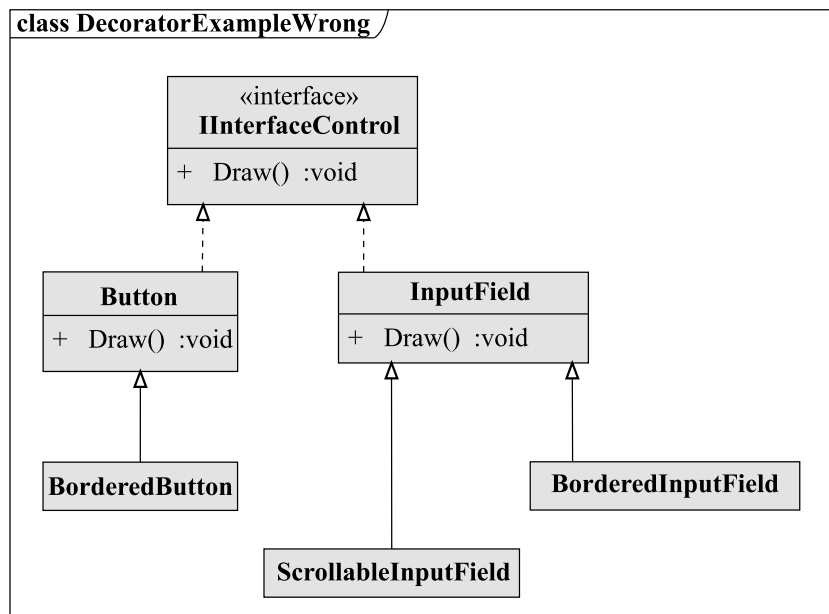


Рис. 7.18 – UML-диаграмма иерархии наследования расширенных элементов управления

В системе появились такие сущности, как *BorderedInputField* и *ScrollableInputField*. Однако если понадобится поле ввода с прокруткой и рамкой, то в систему придется добавить еще один класс *BorderedScrollTextView*. Соответственно, для каждого расширения функциональности придется породить новый класс, а при желании расширить один класс несколькими обязанностями количество классов будет комбинаторно расти.

При использовании паттерна *Decorator* архитектура значительно изменится (рис. 7.19).

В системе появились сущности *BorderedView* и *ScrollView*, которые могут быть применены к любому из имеющихся элементов управления. Причем это может быть сделано на этапе выполнения программы.

Пример добавления обязанностей объектам:

```

//Текст в рамке
var borderedTextView = new BorderedView(new InputField());

//Текст с полосой прокрутки
var scrollTextView = new ScrollView(new InputField ());

//Текст с полосой прокрутки и рамкой
var scrollBorderedTextView = new ScrollView(new BorderedView
(new InputField ()));

//Текст с полосой прокрутки и двумя рамками,
//одной внутри прокрутки и одной общей
var scrollDoubleBorderedTextView = new BorderedView
(new ScrollView(new BorderedView(new InputField ( ))));
  
```

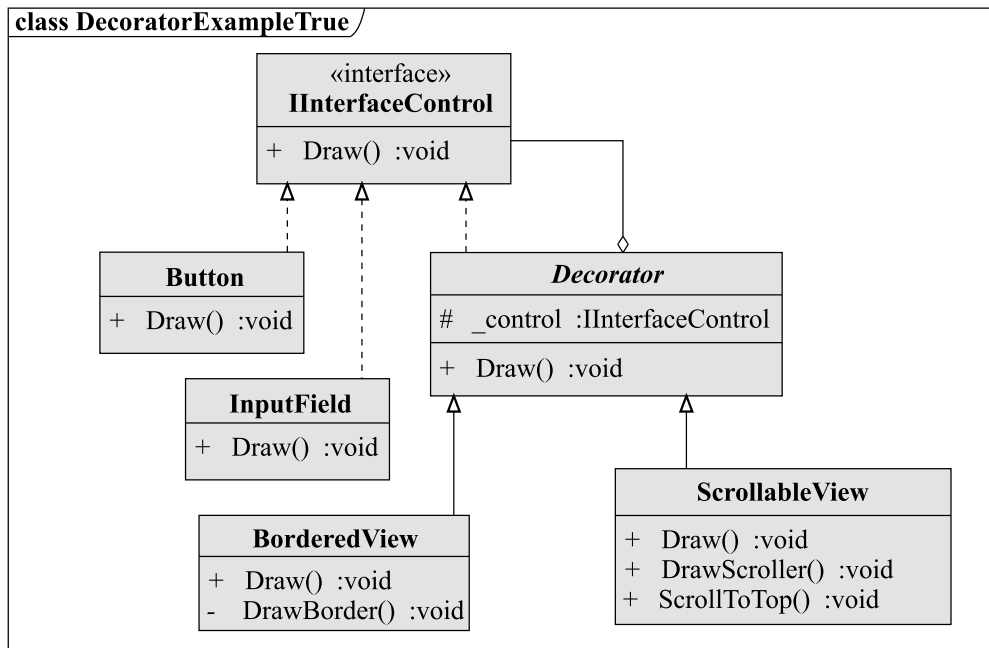


Рис. 7.19 – UML-диаграмма иерархии наследования расширенных элементов управления с применением паттерна *Декоратор*

### Фасад.

*Facade* (Facade) – структурный шаблон проектирования, позволяющий скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, который в свою очередь перенаправляет их соответствующим объектам системы.

Шаблон используется если необходимо:

- упростить доступ к сложной системе;
- создать различные уровни доступа к системе;
- уменьшить число зависимостей между системой и клиентом.

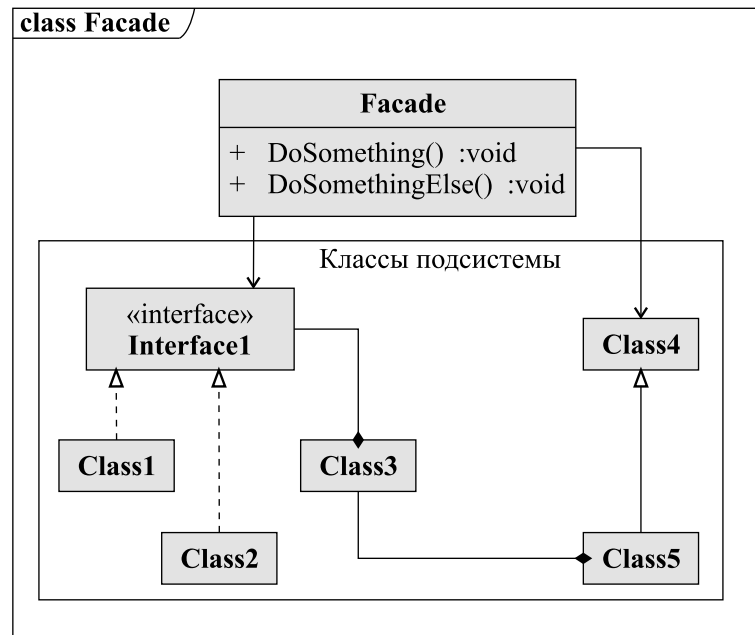
Прежде всего необходимо уточнить, что интерфейс, который предоставляет шаблон, не является суммой всех методов объектов, входящих в систему. Создание такой обобщенной версии приведет к появлению «божественного интерфейса». То есть интерфейса с огромным числом методов, без четко выраженной цели и порождающего большое количество зависимостей. В итоге – прямо противоположный шаблону результат.

Назначение паттерна *Facade* – создать интерфейс, содержащий методы для решения определённой задачи или предоставляющий определённую абстракцию исходной системы. Из этого следует, что одной и той же системе может соответствовать несколько фасадов, решающих разные задачи (рис. 7.20).



### Пример

Классическим примером использования паттерна *Facade* является система компиляции языка ObjectWorks/SmallTalk.

Рис. 7.20 – Схема паттерна *Фасад*

На рисунке 7.21 представлена UML-диаграмма. Как видно, класс *Compiler* предоставляет единую точку входа для системы и через него осуществляется доступ к таким частям системы, как сканер кода, синтаксический анализатор, система представления кода в виде дерева, а также генератор кода.

#### Приспособленец.

*Приспособленец* (Flyweight) – структурный шаблон проектирования, при котором объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту, не является таковым (рис. 7.22). Целью паттерна в данном случае является уменьшение количества экземпляров определенного класса и, следовательно, экономия ресурсов.

Шаблон применяется, если:

- в приложении используется большое число очень схожих экземпляров заданного класса;
- часть состояния объекта является контекстной и может быть легко вынесена во внешние структуры;
- после вынесения части состояния все экземпляры становятся одинаковыми и это дает возможность заменить их одним;
- приложение не проверяет идентичность объектов, т.к. в этом случае все якобы самостоятельные экземпляры являются одним объектом.

Основной сложностью при реализации шаблона является правильное разделение состояния объекта на внутреннее и внешнее. Внешнее состояние передается клиентом, использующим *Приспособленца*, в некотором контексте. Внутреннее состояние хранится непосредственно в *Приспособленце* и позволяет разделять их. Под разделением понимается возможность одновременной работы нескольких клиентов с одним и тем же приспособленцем. Именно возможность сделать это корректно и является главным фактором применимости шаблона.

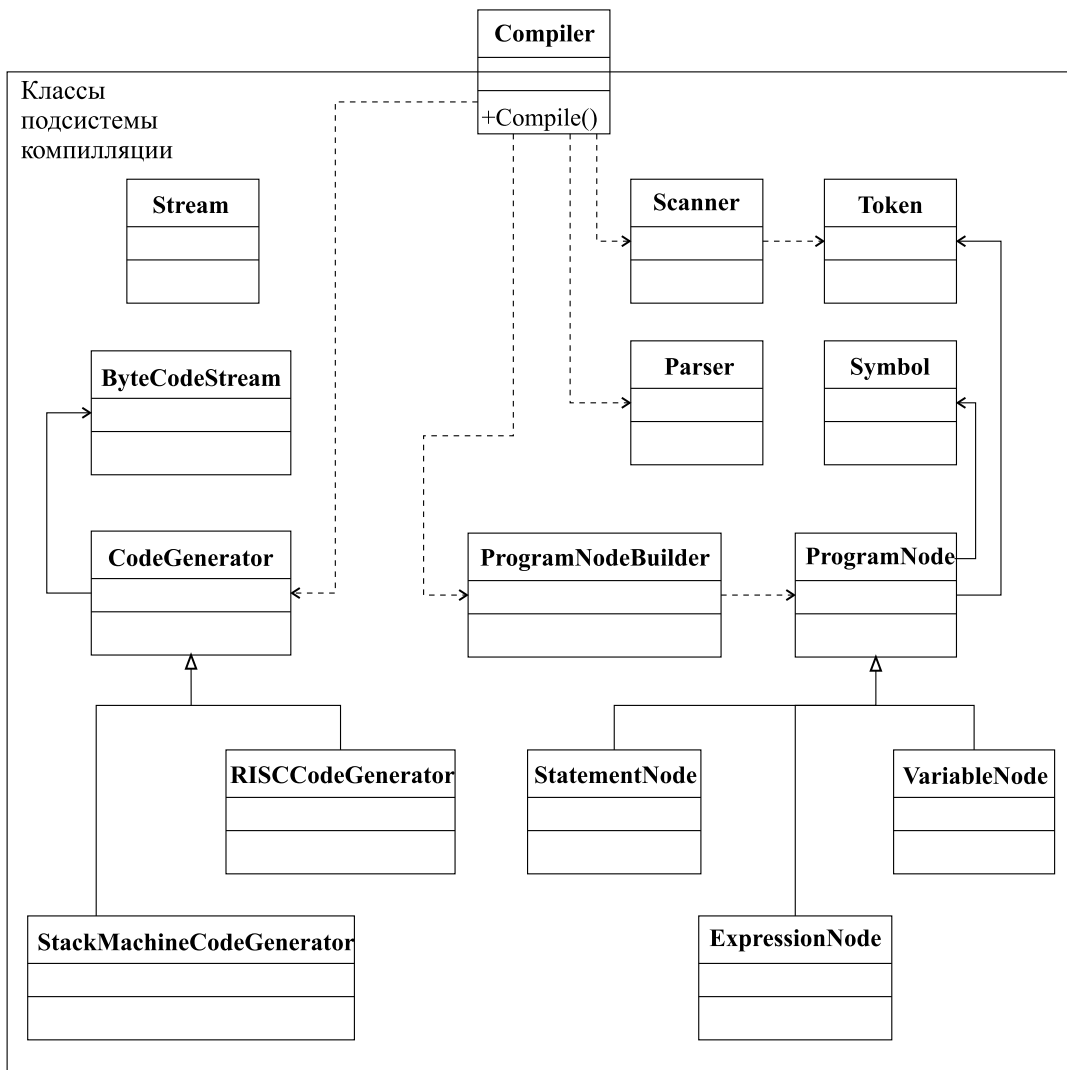
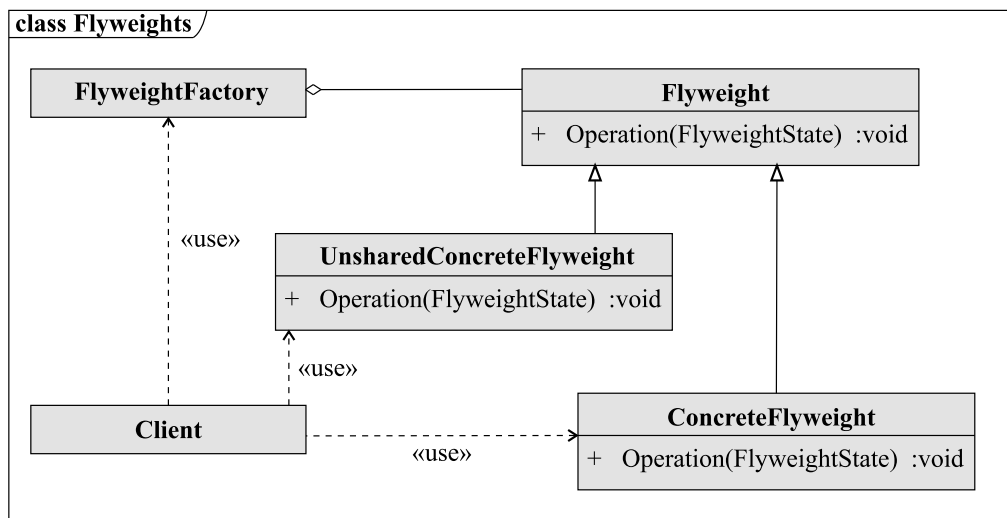


Рис. 7.21 – Диаграмма система компиляции языка ObjectWorks/SmallTalk

Рис. 7.22 – UML-диаграмма паттерна *Приспособленец*



Наличие различных вариантов внутреннего состояния приводит к необходимости создания Пула приспособленцев. При запросе нового экземпляра от клиента, он ищет подходящий вариант среди ранее созданных. Если такого не находится, то порождается новый.



## Пример

Необходимо разработать систему отображения электрических схем. В данной системе существует ограниченное число примитивов — элементов, отвечающих за базовые электрические элементы (резистор, конденсатор, индуктивность). При этом в одной схеме возможно использовать большое число объектов каждого из этих типов.

На рисунке 7.23 представлена диаграмма такой системы.

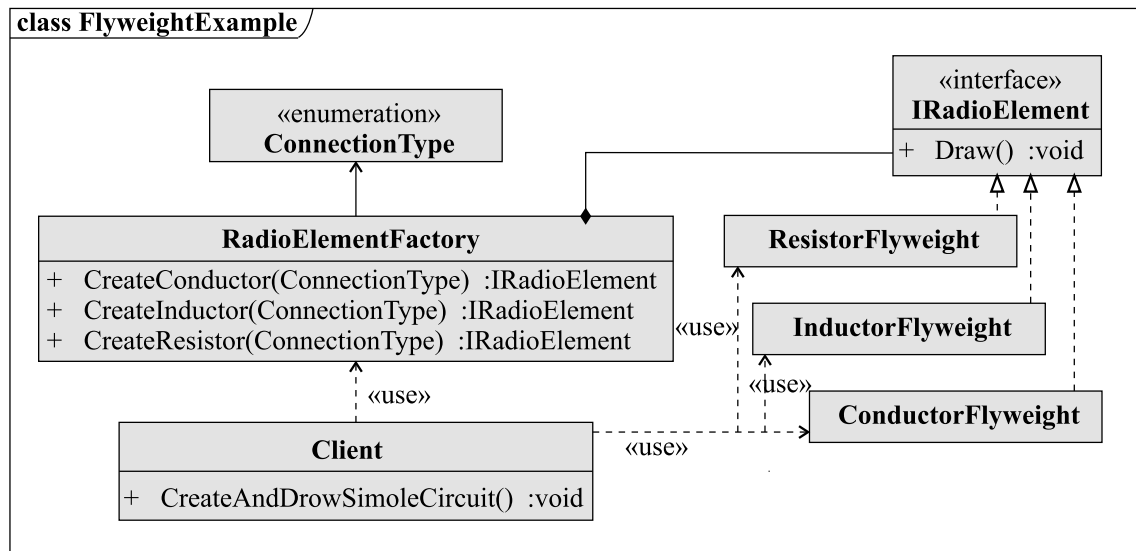


Рис. 7.23 – UML-диаграмма системы отображения электрических схем

В данном случае созданием объектов и их управлением занимается фабрика.

```

class RadioElementFactory
{
    public enum ConnectionType {Series, Parallel};
    private Dictionary<ConnectionType, IRadioElement>
    _resistorMap = new
    Dictionary<ConnectionType, IRadioElement>();
    private Dictionary<ConnectionType, IRadioElement>
    _condensatorMap = new
    Dictionary<ConnectionType, IRadioElement>();
    private Dictionary<ConnectionType, IRadioElement>
    _inductorMap = new
    Dictionary<ConnectionType, IRadioElement>();
}
  
```

```

    public IRadioElement CreateResistor(ConnectionType
connectionType)
    {
        if (_resistorMap.Contains(connectionType))
            return resistorMap[connectionType];
        var resistorFlyweight = new ResistorFlyweight()
        {
            ConnectionType = connectionType
        };
        _resistorMap.Add(connectionType, resistorFlyweight);
        return resistorFlyweight;
    }
    // Аналогичная реализация для конденсаторов
    // и индуктивностей
    ...
}

```

Ниже представлен код приспособленца для резистора, приспособленцы для конденсатора и индуктивности реализованы аналогично.

```

public class ResistorFlyweight : IRadioElement
{
    public ConnectionType ConnectionType{get;set;}
    public void Draw(Point coords)
    {
        //Код отрисовки резистора
    }
}

```

Код создания и отрисовки схемы:

```

public void CreateAndDrawSimpleCircuit()
{
    var circuit = new Circuit();
    circuit.Add(RadioElementFactory.CreateResistor(
RadioElementFactory.ConnectionType.Series));
    circuit.Add(RadioElementFactory.CreateInductor(
RadioElementFactory.ConnectionType.Series));
    circuit.Add(RadioElementFactory.CreateResistor(
RadioElementFactory.ConnectionType.Series));
    circuit.Add(RadioElementFactory.CreateResistor(
RadioElementFactory.ConnectionType.Series));
    circuit.Add(RadioElementFactory.CreateResistor(
RadioElementFactory.ConnectionType.Parallel));
    circuit.Add(RadioElementFactory.CreateInductor(
RadioElementFactory.ConnectionType.Series));
    circuit.Add(RadioElementFactory.CreateCapacitor(
RadioElementFactory.ConnectionType.Parallel));
    circuit.Draw();
}

```

В данной схеме используется четыре резистора, один включённый параллельно и три последовательно, но при этом используются только два экземпляра класса — один для последовательно включенного резистора и один для параллельного.

### Заместитель.

*Заместитель* (Proxy) — структурный шаблон проектирования, который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы (выполняет функцию контейнера). Паттерн позволяет управлять доступом к объекту таким образом, чтобы создавать громоздкие объекты «по требованию» (рис. 7.24).

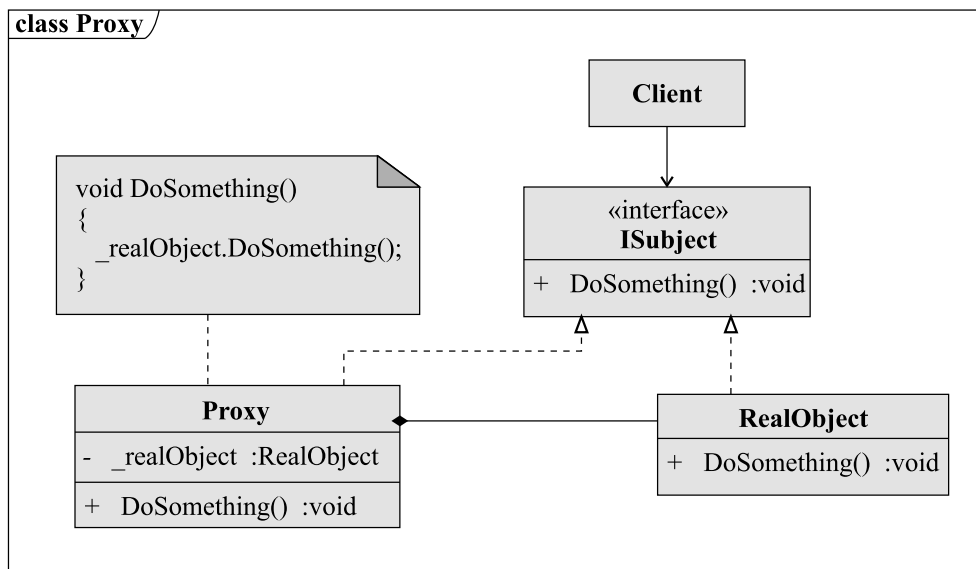


Рис. 7.24 – UML-диаграмма паттерна *Заместитель*

Поскольку интерфейсы реального объекта и объекта-заместителя одинаковые, то клиент может использовать заместитель как реальный объект, а в случае, когда понадобится реальный объект, заместитель может его создать, а затем перенаправить все запросы реальному объекту.

Преимущества данного паттерна:

- удаленный заместитель — предоставляет локального заместителя, вместо объекта, находящегося в другом адресном пространстве или в удаленной машине (удаленный вызов процедур);
- виртуальный заместитель может выполнять оптимизацию — создание объектов по требованию;
- защищающий заместитель — контролирует доступ к исходному объекту (при наличии различных прав доступа к одному объекту);
- «умная» ссылка.



## Пример

Необходимо создать текстовый редактор с возможностью вставки изображений. Загрузка изображения — ресурсоемкая задача, соответственно, необходимо загружать изображение только тогда, когда его необходимо отобразить.

Архитектура данного решения представлена на рисунке 7.25.

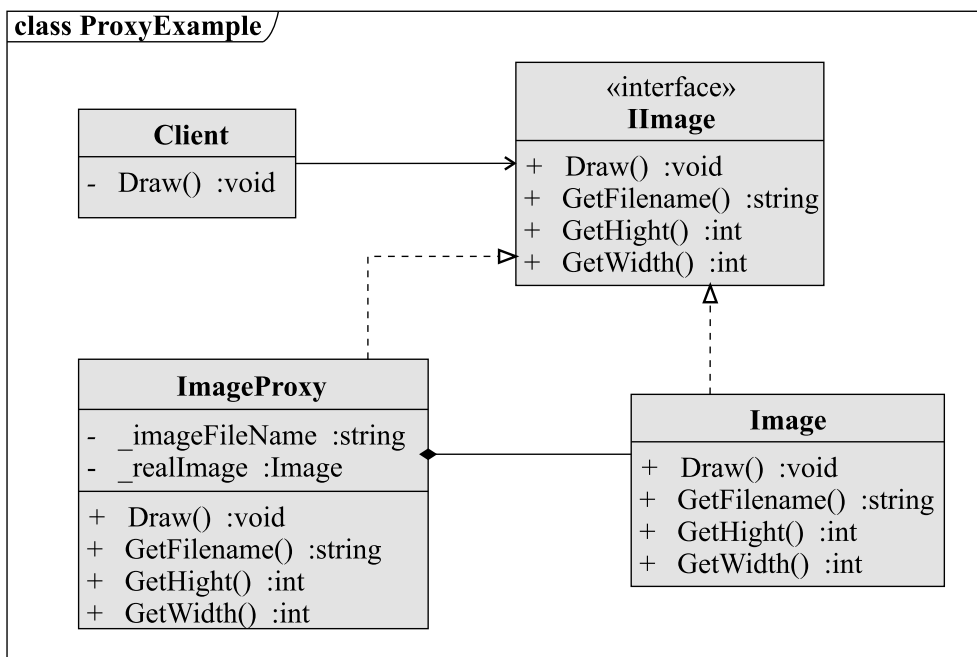


Рис. 7.25 – UML-диаграмма текстового редактора с отложенной загрузкой изображений

Ниже представлена реализация *Заместителя* для объекта *Image*:

```

public class ImageProxy : IImage
{
    private Image _realImage;
    private string _imageFileName;
    public ImageProxy(string filename)
    {
        _imageFileName = filename;
    }
    public string GetFilename()
    {
        return _imageFileName;
    }
    public void Draw()
    {

```

```
        if (_realImage == null)
            _realImage = Image.LoadFromFile(_filename);
        _realImage.Draw();
    }
    public int GetWidth()
    {
        if (_realImage == null)
            _realImage = Image.LoadfromFile(_filename);
        return _realImage.Width;
    }
    public int GetHeigth()
    {
        if (_realImage == null)
            _realImage = Image.LoadfromFile(_filename);
        return _realImage.Heigth;
    }
}
private void Draw()
{
    var img = new ImageProxy("testFile");
    // картинка не загрузилась
    //
    // Произвольный код
    //
    var filename = img.GetFilename();
    // картинка не загрузилась, так как имя хранится
    // в заместителе
    var width = img.GetWidth();
    // необходимо загрузить картинку
    img.Draw();
    // Отрисовка изображения (делеглируем классу Image)
}
```

## 7.2 Антипаттерны

В предыдущей части были рассмотрены паттерны проектирования — примеры стандартных решений часто встречающихся задач. Их использование позволяет повысить гибкость, расширяемость ПО. Антипаттерны — полная противоположность. Это типичные ошибки, с которыми встречается каждый разработчик, и для того чтобы считаться хорошим программистом, надо уметь избегать таких ошибок.

### **Программирование методом копирования и вставки.**

Одна из самых часто встречающихся ошибок начинающих программистов — программирование методом копирования и вставки, на программистском сленге «копипаст» (от английского *copy&paste*). Возникает, когда требуется написать две похожие функции. Программист, вместо создания общего решения, копирует уже

созданную функцию и вносит небольшие изменения. Такой подход ведет к следующим проблемам:

- ухудшается переносимость кода — если потребуется подобный функционал в другом проекте, то надо будет искать все места, где программист «накопипастил», и переносить их по отдельности;
- понижается качество кода — часто программист забывает вносить требуемые изменения в скопированный код;
- усложняется поддержка кода — если в изначальном варианте был баг, который в будущем надо будет исправить, то этот баг попал во все методы, которые «накопипастил» программист. Значительно затрудняет отладку и исправление ошибок программы.



.....  
 Данная ошибка считается одной из самых опасных. Часто возникает из-за неопытности программиста или нежелания прогнозировать возможные изменения в логике работы программы.  
 .....

#### Спагетти-код.

Под *спагетти-кодом* подразумевают слабо структурированную программу с большим количеством связей между объектами, что приводит к сложности понимания логики работы программы для других программистов. Соответственно это ведет к сложности модификации программы, ее расширения. Часто такой код содержит множество проявлений предыдущего антипаттерна, огромные классы (по несколько тысяч строк кода) с огромными же методами. Причина возникновения — недостаток опыта программиста или использование принципа «лишь бы работало».



.....  
 В программах или модулях, в которых встречается спагетти-код, необходимо проводить рефакторинг или даже полностью переписывать, это оправдывает себя в будущем даже при временных затратах.  
 .....

#### Золотой молоток.

*Золотой молоток* — уверенность в универсальности какого-либо решения. Часто это использование одного и того же паттерна или подхода для решения всех встречающихся задач, даже если это неуместно или в этом нет необходимости. Данную ошибку совершают как неопытные программисты, так и опытные, которые считают, что овладели каким-либо паттерном в совершенстве. Решение проблемы — анализ системы и поиск нескольких решений. Обычно систему можно спроектировать несколькими различными способами, и хороший программист должен рассмотреть их все и выбрать лучший подход для решения текущей задачи.

#### Магические числа.

*Магическое число* — какая-либо неименованная константа, используемая в коде. Проблема в том, что число не несет никакого смысла без комментария, в результате, программист, который не сам писал текущий модуль, не сможет понять, что

обозначает то или иное число. Результат — сложность понимания работы программы и сложности в ее расширении.



.....  
 Решение данной проблемы — внесение константы в виде поля класса либо использование перечислений, если это возможно.  
 .....

### **Жесткое кодирование.**

*Жесткое кодирование* — использование в коде данных об окружении. Например, использование абсолютных путей файлов. Часто встречается в паре с предыдущим паттерном. Очевидна проблема — невозможность переноса программы, потому что, например, на другом устройстве необходимые файлы будут расположены по другому пути.

### **Мягкое кодирование.**

*Мягкое программирование* — написание программы, в которой абсолютно все может быть или должно быть настроено пользователем. Противоположность жесткого программирования. Такой подход ведет к чрезмерной сложности настройки программы пользователем. Также это ведет к трате ресурсов на реализацию настроек всего при разработке.



.....  
 Для решения этой проблемы необходимо перед началом разработки определить, какие параметры должны настраиваться пользователем, а какие должны быть постоянными или могут настраиваться автоматически.  
 .....

### **Ненужная сложность.**

*Ненужная сложность* — ошибка, при которой типичная задача решается сложным, часто неочевидным путем. При этом такое решение не несет никакой выгоды. Внести излишнюю сложность можно в любую задачу, и надо стараться избегать этого, потому что это ведет к непрозрачности кода, затруднению его понимания.



.....  
 Решение данной проблемы — рефакторинг (или переписывание) неоправданно сложного участка.  
 .....

### **Лодочный якорь.**

Этот антипаттерн подразумевает сохранение неиспользуемых частей кода, которые остались после рефакторинга или оптимизации. Такой код только усложняет систему, не неся никакой практической ценности.

### **Изобретение велосипеда.**

Под этим понимается разработка своего решения типичных задач, при наличии уже существующих решений, которые зачастую лучше. Проблема в том, что программист считает себя лучше других и думает, что сможет сам придумать наилучшее решение для любой задачи. Обычно это ведет к банальной потере времени.

Однако стоит сохранять баланс и не отказываться от разработки своих решений. Разработчик должен уметь анализировать поставленные задачи и делать выбор между использованием готового решения, либо разработкой своего.

#### **Изобретение одноколесного велосипеда.**

Развитие предыдущего антипаттерна — разработка своего худшего решения при существовании лучшего. Также ведет к потере времени, в частности на последующий рефакторинг и переписывание программы для использования более оптимального решения.

#### **Программирование перебором.**

Многие начинающие программисты пытаются решать некоторые задачи методом перебора, подбором параметров, изменением порядка вызова функций. Такой подход может обеспечить работоспособность программы, но не дает программисту понимания сути происходящего, следовательно, программист не сможет в будущем предусмотреть все возможные варианты развития событий. Программист потратит много времени на подбор параметров для текущего решения и впоследствии потратит столько же времени на переписывания этого решения для других нужд. Если программист ищет решения методом перебора, это характеризует его, как ленивого человека либо как человека, который не способен аналитически мыслить.

#### **Слепая вера.**

Суть данного паттерна в недостаточной проверке входных параметров. Очень часто программисты, особенно начинающие, считают, что их программа будет работать в идеальных условиях, что, конечно же, неверно. Однако это не значит, что необходимо проверять все, такой подход приведет к излишней сложности, достаточно проверять все данные, которые может ввести пользователь, и возможные проблемы в чужом коде, который использует программист.

#### **Бездумное комментирование.**

Название данного антипаттерна говорит само за себя. Не стоит комментировать все только ради наличия комментария, потому что комментарии призваны упростить понимание кода, а множество лишних комментариев может только затруднить понимание.



.....  
 Следует комментировать не *что* происходит в определённом участке кода, а *почему* вы решили сделать именно так.  
 .....

Также следует избегать диалога разработчиков в комментариях, для этого существуют отдельные инструменты.

#### **Божественный объект.**

*Божественный объект* — объект, который берет на себя слишком много обязанностей, хранит в себе множество данных и методов. Результатом является код, который сложно переносить и достаточно сложно поддерживать, потому что вся система зависит практически только от него. Причина возникновения данного антипаттерна — недостаток опыта разработчика, неспособность проводить грамотную декомпозицию системы.



**Поток лавы.**

Данный антипаттерн подразумевает сохранение нежелательного (излишнего или низкокачественного) кода по причине того, что его удаление слишком дорого или будет иметь непредсказуемые последствия. Последствия очевидны — наличие в программе участков кода, которые непонятны, неочевидны, скорее всего, являются запутанными, соответственно возникают проблемы при переносе и поддержке данного кода, а самое главное, что добавление функциональности в такие участки чаще всего либо невозможно, либо ведет к накоплению некачественного кода в программе.



.....  
 Решением данной проблемы является переписывание некачественного участка.  
 .....

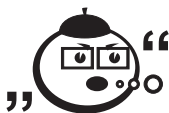
## 7.3 Оформление кода

В предыдущих частях было рассмотрено, как следует и как не следует разрабатывать ПО с точки зрения архитектуры, то есть были даны типичные ошибки и решения типичных задач проектирования. Эти знания позволяют сделать архитектуру программы более простой и понятной, и, следовательно, её будет проще поддерживать, расширять. Однако не только правильная архитектура определяет понятность программы, какая бы она ни была, невозможно разобраться в программе, если ее код написан без какого-либо оформления.



.....  
 Статистика показывает, что любой программист проводит больше времени за чтением кода, чем за его написанием. Читать код намного сложнее, чем писать, особенно, если его написал другой программист. Часто программист сам не может разобраться в своем коде годичной или даже месячной давности.  
 .....

Именно поэтому программист должен обеспечить читаемость программного кода, который он пишет. М. Фаулер, большой специалист в написании и поддержке кода, писал, что:



.....  
 «Любой дурак может написать код, понятный компьютеру. Хорошие программисты пишут код, понятный людям».  
 .....



.....  
 В среде программистов уже давно существует такое понятие, как **стандарт оформления кода**, — набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования.  
 .....

Наличие общего стиля программирования позволяет облегчить понимание и поддержку исходного кода, а также упрощает взаимодействие нескольких человек, работающих над программой. В идеале, стандарт оформления кода имеет перед собой цель добиться такого состояния кода, что любой достаточно квалифицированный программист с одного взгляда может определить, что делает та или иная функция, переменная, каким образом функционирует определенный участок кода и, возможно, даже определить, корректно ли данный код работает. Поэтому стандарт кодирования обычно строится так, чтобы за счёт определённого визуального оформления элементов программы повысить информативность кода для человека.

Следует понимать, что не существует универсального стандарта оформления, каждый стандарт разрабатывается для конкретного языка, таким образом, стандарт для C# будет значительно отличаться от стандарта для Java. В последнее время также наблюдается тенденция включения правил оформления кода в качестве необходимого элемента синтаксиса, например в Python, логика выполнения блоков зависит от отступов (то, что в других языках определялось либо скобками, либо ключевыми словами). То есть если раньше программист оформлял код по своему желанию (или не оформлял), то теперь синтаксис языка сам диктует необходимость правильного оформления кода.

Как уже говорилось, стандарт разрабатывается под каждый конкретный язык программирования, но у всех есть общие требования.

1. Именованние классов, интерфейсов, полей классов, методов, переменных.

Под этим подразумевается два требования:

- осмысленность — имя переменной должно явно указывать на ее назначение, название метода должно явно указывать, какое действие данный метод выполняет, и так далее. Такое простое требование позволит впоследствии значительно упростить понимание логики работы и позволит находить некорректные с точки зрения логики места, например, если в названии метода присутствует два действия (например, `DecodeAndSendData`), то этот метод надо разделить, потому что каждый метод должен решать одну задачу. Это один из ярких примеров преимущества осмысленного именования. Также необходимо избегать сокращений и аббревиатур, кроме общеупотребительных (например, API или SDK);
  - стиль именования — нижний регистр, ВЕРХНИЙ РЕГИСТР, СтилПаскаль, стильКамел, а также наличие или отсутствие символов «`_`» в имени. Обычно для различных типов, допустим, переменных (локальные, аргументы, поля класса), применяется отличный стиль именования — это позволяет только по названию переменной определить, переменной какого типа она является.
2. Стил отступов для логических блоков. Здесь применяется табуляция или пробел, а также указывается размер отступа. Это позволяет сразу определить, какой блок кода, допустим, относится к циклу или попадает под определенное условие.
3. Способ расстановки скобок, ограничивающих логические блоки.

4. Использование пробелов при оформлении логических и арифметических выражений.
5. Стиль комментариев, используемых при описании кода.

Также вне стандарта подразумеваются отсутствие «магических чисел» и размеры методов по горизонтали и вертикали. Конечно, не существует количественных ограничений, просто считается, что метод должен вмещаться на один экран (для его просмотра не надо прокручивать экран по горизонтали или по вертикали).

Следует отметить, что в настоящее время любая компания, которая занимается разработкой ПО, либо имеет свой стандарт оформления, либо использует уже существующий. И его использование является обязательным требованием при разработке, поэтому первое, что должен сделать программист, который только что устроился на работу в компанию по разработке ПО, — узнать стандарт оформления кода, применяемый в этой компании.

## 7.4 Рецензирование кода



.....  
*Рецензирование кода, или код ревью (от английского code review), — инженерная практика, применяемая в рамках гибких методологий разработки, подразумевает инспекцию кода с целью выявления ошибок, недочетов, расхождения в стиле написания кода и соответствии написанного кода и поставленной задачи.*  
 .....

Данная практика обладает следующими преимуществами:

- нахождение ошибок (опечаток в коде). Рецензирование помогает находить и исправлять ошибки быстрее, поскольку проводится на ранних этапах разработки (относительно формального тестирования);
- повышается степень совместного владения кодом. На рецензирование разработчики узнают о коде и логике других частей проекта, это позволит впоследствии облегчить переход разработчика от одной части проекта к другой;
- происходит обмен опытом между разработчиками, полезные практики одного разработчика перенимаются другими;
- улучшается качество кода, код приводится к единому стилю написания;
- шаг к грамотному рефакторингу.

Обычно рецензирование кода проводится либо периодически (например, раз в неделю), либо разработчик проводит рецензирование после завершения некоторой объемной задачи (например, написание определенного модуля программы). В принципе можно проводить рецензирование любого кода, но, допустим, проведение рецензирования простой интерфейсной формы, по большому счету, трата времени. Обязательно необходимо проводить рецензирование критических мест в приложении (механизмы аутентификации, авторизации, передачи и обработки важной информации — обработка денежных транзакций). Часто проводят так назы-

ваемые тематические рецензирования — частный случай рецензирования кода, проверка некоторого модуля на соответствие некоторой цели. Важным в этом случае является четкая постановка цели задачи. Преимуществом такого рецензирования является то, что рецензируется достаточно небольшой участок кода, что позволяет глубоко в него вникнуть и дать ценные замечания.

Вообще рецензирование состоит из двух элементов: *design review* и *code review*.

*Design review* — анализ будущей архитектуры программы. Этот этап очень важен, так как без него рецензирование будет менее полезным или вовсе бесполезным, потому что написанный программистом код полностью неверен, поскольку не решает поставленную задачу либо не удовлетворяет каким-либо требованиям (по времени выполнения, по памяти и т. д.). На этом этапе тот, кто будет писать код, рассказывает о общей стратегии, приводит UML-диаграммы, примерные алгоритмы, инструменты, используемые библиотеки и т. д. Проведение данного этапа позволяет значительно экономить время, потому что ошибка проектирования выявляется сразу, а не после того, как код написан и выясняется, что он не удовлетворяет поставленным требованиям.

*Code review* — непосредственно рецензирование кода, на данном этапе автору отправляются пожелания и замечания по логике и оформлению кода.

Проводить рецензирование можно несколькими способами, каждый из которых имеет свои достоинства и недостатки. Первый способ — когда команда собирается перед монитором одного из коллег либо перед проектором и начинает рецензировать код. Преимуществом данного подхода можно назвать то, что разработчик может комментировать ход своих мыслей по ходу во время написания, по ходу рецензирования. Так, разработчик может доказать, что его, не совсем очевидный, подход, является лучшим, и тогда результатом рецензирования будут лишь замечания по оформлению и дополнительному комментированию неочевидного участка кода, а не его полное переписывание. С другой стороны, такой подход отнимает много времени и рецензенты не всегда успевают вдуматься в код и дать полезные замечания по коду. Опять же, таким рецензированием надо очень строго управлять, потому что оно может вылиться в бурное обсуждение или в рассказы веселых историй из жизни рецензентов и т. д. Вторым подходом является дистанционное рецензирование. При таком подходе всем в команде передается код для рецензии, и каждый уже может выбрать удобное ему время и вдумчиво изучить код и выдать веские замечания. Однако при таком подходе может значительно затянуться получение обратной связи, потому что разработчики могут сослаться на недостаток времени и откладывать рецензирование.

Результатами рецензирования могут являться:

- описание способа решения задачи (*design review*);
- UML-диаграммы (*design review*);
- более правильный (быстрый, легко читаемый) вариант реализации (*design review, code review*);
- комментарии к стилю кода (*code review*);
- указание на ошибки в коде (*code review*);
- юнит-тесты (*design review, code review*).

При этом очень важно, чтобы результаты не терялись, поэтому они вносятся в базу знаний проекта или в систему управления проектами. Затем следует этап рефакторинга (см. ниже), на котором данные замечания будут обработаны и еще раз отправлены на рецензирование.

## 7.5 Рефакторинг

Разработка ПО процесс итерационный, невозможно сразу написать код, который бы соответствовал всем стандартам, работал без ошибок, хорошо читался, легко модифицировался, обладал хорошей скоростью работы. Кроме того, часто программист в первую очередь добивается работы необходимой функциональности, не обращая внимания (или обращая в меньшей степени) на его оформление, комментирование и т. д. Однако, со временем, программист замечает, что модуль можно было спроектировать иначе, либо метод можно написать более понятно и наглядно, либо ему необходимо добавить новую функциональность, но не может понять тот код, который необходимо модифицировать. Если программист сталкивается с чем-то подобным, значит, пришло время для рефакторинга.



.....  
**Рефакторинг** (англ. *refactoring*) или *реорганизация кода* [52] — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы.  
 .....

Обычно рефакторинг подразумевает внесение небольших изменений в программу, каждое из которых не затрагивает логику поведения программы, но упрощает или улучшает код. При этом очень важно, чтобы изменения проводились небольшими частями, потому что за правильностью таких изменений проще следить. Однако не стоит думать, что рефакторинг это банальное переименование переменных или комментирование кода, рефакторинг позволяет полностью переработать программу, улучшить ее согласованность, главное — вносить изменения маленькими частями.



.....  
 Чтобы не испортить уже существующий функционал, обычно рефакторинг проводится при наличии средств контроля за функционалом, которыми обычно выступают модульные тесты.  
 .....

Следует отличать рефакторинг и оптимизацию кода, потому что главная цель рефакторинга — улучшить качество и понятность кода, а цель оптимизации — ускорить его работу, при этом, часто, ускорение работы влечет значительное ухудшение понятности кода.

Можно выделить следующие преимущества проведения рефакторинга:

- рефакторинг улучшает архитектуру программного обеспечения. По мере внесения в программу изменений, связанных с добавлением новой функ-

циональности, без полного понимания архитектуры программы, она приходит в негодность. Становится все сложнее разбираться в проекте. Рефакторинг позволяет навести порядок в коде, убрать ненужные фрагменты, перенести некоторые фрагменты в другие классы, выделять более удобные интерфейсы и т. д.;

- рефакторинг облегчает понимание кода. Как уже говорилось, часто программист, реализуя функциональность, не обращает внимание на читаемость кода. Поэтому при рефакторинге вы берете работающий код и делаете его более читабельным. Это важно, потому что впоследствии, возможно, этим проектом будет заниматься кто-то другой и понятный код позволит ему быстрее приступить к внесению изменений и исключить значительные временные затраты на понимание работы кода;
- рефакторинг позволяет найти ошибки. Лучшее понимание кода помогает находить ошибки. При рефакторинге программисту необходимо вдумываться в то, что делает код, его логику. При подобном глубоком понимании ошибки работы кода «всплывают» сами;
- рефакторинг позволяет быстрее писать программы. Все вышесказанное приводит к этому пункту, потому что чем четче архитектура программы, чем проще разбираться в логике работы кода и чем меньше тратится времени на поиск ошибок, тем быстрее идет разработка.

Периодически любой код необходимо рефакторить, при этом необходимо, в основном, полагаться на опыт и интуицию. Если вам не понятно, как работает метод, или вы бы написали этот метод иначе, значит, его необходимо рефакторить. Можно выделить основные признаки того, что необходимо провести рефакторинг:

- В программе есть дублирование кода. О недостатках дублирования уже говорилось в главе про антипаттерны (Программирование методом копи-паста), поэтому не будем повторяться. Данная проблема решается выносом дублированного кода в отдельный метод.
- В программе есть очень длинные методы. Как правило, человек не может полностью воспринимать и оценивать правильность кода, если этот код занимает больше 2–3 десятков строк. Такие методы и функции следует разделять на несколько более мелких и делать одну общую функцию, которая будет последовательно вызывать эти методы. Не стоит пытаться сократить длину метода, записывая по несколько операторов в одной строке, — это один из самых худших вариантов организации метода, такой код в итоге приведёт к ошибке.
- Длинный список параметров метода/конструктора. Большое количество параметров обычно не только усложняет понимание того, что делает этот метод, но и усложняет понимание кода, использующего эти функции. Если действительно необходимо, чтобы функция принимала очень много параметров, — вынесите эти параметры в отдельную структуру (либо класс), дав этой структуре понятное имя, и используйте ее в качестве параметра метода.
- Большие классы так же требуют рефакторинга. Если в программе есть один или несколько больших классов, их следует немедленно разделить их на

более мелкие и включить объекты этих классов в один общий класс. Причина этого та же самая, что и в предыдущем пункте.

- Слишком много временных переменных так же являются признаком плохого кода. Как правило, много временных переменных встречаются в излишне «раздутых» функциях — после проведения рефакторинга скорее всего количество временных переменных станет меньше и код станет значительно понятнее и удобнее.
- Много «беспорядочно» хранящихся данных, которые связаны логически. Такие данные необходимо объединять в структуры либо классы, даже если это всего 2–3 переменные.
- Если объекты одного класса слишком часто обращаются к данным другого объекта — следует пересмотреть функционал объектов. Возможно, было принято неверное архитектурное решение и его надо поменять как можно раньше, пока эта ошибка не распространилась по всему коду.
- В системе слишком много глобальных переменных. Если множество глобальных переменных действительно необходимо — можно сгруппировать их в структуры/классы, либо вынести в отдельное пространство имен, тогда шанс того, что какая-либо переменная будет использована случайно, по ошибке, станет значительно ниже. Также можно использовать паттерн *Одиночка*.



.....

В заключение можно сказать, что рефакторинг необходимо проводить всегда, а не откладывать его на потом, потому что час, затраченный на рефакторинг сегодня, может сэкономить значительно больше времени при поиске ошибок. У программиста должно войти в привычку, что при внесении любых изменений в коде, добавлении нового метода, класса и т. д. надо посмотреть на соседний код и подумать, можно ли внести в него какие-либо изменения, которые позволят его улучшить.

.....

## 7.6 Оптимизация

Раньше, когда у компьютеров были килобайты оперативной памяти, мегабайты дискового пространства и частоты процессора исчислялись мегагерцами, требования к программам были намного жестче, программистам приходилось намного скрупулёзней подходить к написанию программ, потому что каждый дополнительный мегабайт используемой оперативной памяти был «непозволительной роскошью» и мог значительно сказаться на эффективности и популярности программы. Программистам приходилось очень тщательно оптимизировать программу. Сейчас мощности компьютеров выросли в разы, и тщательная оптимизация ПО отходит на второй план, уступая место скорости разработки. Однако даже сейчас есть приложения, требующие высокой скорости работы, например приложения по обработке видео, аудио, а также компьютерные игры. Поэтому не надо преуменьшать роль оптимизации в разработке ПО.



.....  
**Оптимизация** — модификация системы для улучшения её эффективности.  
 .....

Хотя целью оптимизации является получение оптимальной системы, истинно оптимальная система в процессе оптимизации достигается далеко не всегда. Оптимизированная система обычно является оптимальной только для одной задачи или группы пользователей: где-то может быть важнее уменьшение времени, требуемого программе для выполнения работы, даже ценой потребления большего объёма памяти; в приложениях, где важнее память, могут выбираться более медленные алгоритмы с меньшими запросами к памяти.

Оптимизация в основном фокусируется на одиночном или повторном времени выполнения, использовании памяти, дискового пространства, пропускной способности или некотором другом ресурсе. Это обычно требует компромиссов — один параметр оптимизируется за счёт других. Например, увеличение размера программного кэша чего-либо улучшает производительность времени выполнения, но также увеличивает потребление памяти. Другие распространённые компромиссы включают прозрачность кода и его выразительность почти всегда ценой деоптимизации. Сложные специализированные алгоритмы требуют больше усилий по отладке и увеличивают вероятность ошибок.

Важно понимать, что оптимизация (как и рефакторинг) начинается, когда уже написана основная функциональность приложения, и цель оптимизации только лишь улучшение какого-либо параметра ПО, без изменения его функциональности. Тони Хоар впервые произнёс, а Дональд Кнут впоследствии часто повторял известное высказывание:



.....  
 «Преждевременная оптимизация — это корень всех бед».  
 .....

Очень важно иметь для начала работающий прототип.

Можно выделить несколько основных принципов оптимизации.

- В первую очередь необходимо проводить оптимизацию самых «узких» мест программы. Узкое место, иногда его еще называют «бутылочное горлышко» — критическая часть кода, которая является основным потребителем необходимого ресурса. Если оптимизировать те места, где и без вмешательства все быстро работает, то прирост производительности будет минимален. Для поиска узких мест используются специальные программы — профайлеры.
- Архитектурный дизайн системы особенно сильно влияет на её производительность. Выбор алгоритма влияет на эффективность больше, чем любой другой элемент дизайна. Более сложные алгоритмы и структуры данных могут хорошо оперировать с большим количеством элементов, в то время как простые алгоритмы подходят для небольших объёмов данных —



накладные расходы на инициализацию более сложного алгоритма могут перевесить выгоду от его использования.

- Оптимизировать лучше те места, которые регулярно повторяются в ходе работы. Этот закон относится к циклам и подпрограммам. Если можно хотя бы немного оптимизировать цикл, то это стоит сделать. Если в одной итерации получится добиться прироста в 2%, то после 1000 повторений это уже будет достаточно большое значение.
- Не стоит злоупотреблять оптимизацией единичных операций. Это вытекает из предыдущего принципа. Оптимизируя фрагмент, который будет вызван лишь один раз, вряд ли получится добиться ощутимого прироста.
- Необходимо задумываться над оптимизацией. Неправильная оптимизация может даже навредить программе, увеличить время ее разработки, практически не уменьшив скорость ее работы.



## Контрольные вопросы по главе 7

1. Что такое паттерны и для чего они нужны?
2. Что такое антипаттерны?
3. Какие проблемы несет программирование методом копипаста?
4. Почему необходимо оформлять код в едином стиле?
5. Что такое рефакторинг? Зачем он нужен?
6. Что такое рецензирование кода, зачем оно используется?

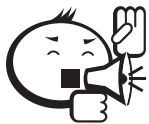
---

## Глава 8

# ТЕСТИРОВАНИЕ

---

Как было отмечено в главе 1, современная разработка программного обеспечения не может обойтись без этапа тестирования. В настоящее время всё большее количество компаний приходит к выводу, что проведение тестирования на всех этапах разработки позволяет значительно сократить расходы. Данный тезис подтверждается и многочисленными исследованиями [53].



.....  
Тестирование предназначено для определения качества продукта.  
Заметьте, *определения*, а не повышения!  
.....

Это значит, что проведение тестирования сможет вам только указать, какие части вашей программы являются качественными, а какие являются проблемными. Однако само тестирование не улучшает качество тех или иных частей программы. Поэтому если в результате тестирования выяснилось, что один из ключевых модулей разрабатываемой программы содержит запредельное количество ошибок, то кто-то из разработчиков должен тут же (ну, или согласно плану разработки) заняться исправлением найденных ошибок. Иначе качество продукта останется таким же отвратительным, как и до проведения тестирования. Если же тестирование не проводится вовсе, то это не делает качество вашей программы заведомо плохим или хорошим. Отсутствие тестирования делает качество вашей *программы неопределенным*, а значит, вы уже не сможете гарантировать вашему пользователю адекватную работу продукта. Меньше всего пользователи любят продукты, которым они не могут доверять. Стали бы вы покупать себе электрический чайник, производитель которого не гарантирует отсутствия самовключений или самовозгораний?

Как показывает статистика, тестирование может занимать четверть, а то и половину от всего объёма работ над программой. Данный факт часто приводит начинающие компании к мысли, что поскольку тестирование само по себе не улучшает

качество продукта, то на данном этапе можно значительно сэкономить — ведь лучше нанять больше программистов, чем «каких-то» тестировщиков. Однако здесь же нужно знать и обратную сторону — отсутствие тестирования увеличивает время разработки в 1.5–2 раза (!) за счет необходимости отладки и поддержки кода. Подробная таблица о том, сколько могут стоить ошибки на различных этапах разработки, также приводится в главе 1. Таким образом, вы можете либо уделить четверть времени при разработке на тестирование, либо вдвое затянуть сроки разработки. Решайте сами.



.....

В тестировании также есть несколько постулатов, которые нужно принять за истину. Первый постулат гласит, что *абсолютно любая программа содержит ошибки!* Это нужно принять как разработчикам, так и тестировщикам. Всегда можно найти нюансы, которые не были учтены при разработке. Второе правило, которое является следствием из первого, — *ни один продукт нельзя протестировать на 100% или нельзя полностью гарантировать качество продукта.* Поэтому разработчик должен понять, что всегда можно улучшить программу, а тестировщик — что у него всегда будет работа (если он хороший тестировщик, разумеется).

.....

## 8.1 Что такое тестирование?



.....

**Тестирование** — это проверка соответствия объекта желаемым критериям.

.....

Несоответствие объекта желаемым критериям называется *ошибкой*. Из этого определения следует, что для проведения тестирования нужны три вещи:

- объект, который нужно протестировать, — собственно, разрабатываемая программа;
- описание того, как этот объект должен себя вести, — техническое задание;
- методы проверки/сравнения объекта с описанием — план тестирования.

Если объект тестирования присутствует в компании с той самой секунды, когда разработчик написал первую строчку кода, то техническое задание существует далеко не всегда, — такое явление часто можно встретить в маленьких начинающих фирмах, нежелающих тратить время на «бумажную возню». Поэтому при организации тестирования первое, что должен сделать тестировщик, — ознакомиться с техническим заданием. Если такого документа нет, тестировщик должен начать его требовать у своего непосредственного руководителя. Аргументация при этом должна быть следующая: «Нельзя протестировать программу без описания того, как программа должна себя вести». Фактически, именно внедрение тестирования в процесс разработки выявляет все недостатки отсутствия проектной документации.

Тестировщик должен протестировать каждую деталь продукта, однако различные составляющие проекта имеют различную важность. И качество покрытия той или иной части тестами должно определяться важностью данной части для всей системы в целом. Для примера, ошибка в модуле обработки изображений в графическом редакторе менее важна, чем аналогичная ошибка в медицинской программе по работе с рентгеновскими снимками. Поэтому модуль обработки изображений в медицинском программном обеспечении должен быть протестирован куда более тщательно, чем модуль обработки изображений в графическом редакторе.

Таким образом, тестировщик должен грамотно распределить своё внимание по продукту. Ключевые элементы программы и основные критерии качества продукта должны быть отражены в техническом задании. Если их нет, данный вопрос необходимо проработать с заказчиком и руководителем проекта. Тестировщик в первую очередь должен обеспечить проверку именно этих критериев качества, так как относительно этих качеств в дальнейшем продукт будет позиционироваться для пользователя и именно благодаря этим качествам пользователь будет приобретать продукт. Но это не значит, что всё, что не перечислено в перечне критериев качества, не должно тестироваться. Тестировщик должен обеспечить максимальное тестирование продукта!

Многие критерии качества сложно оценить, работая со всем объектом целиком. В большинстве случаев проще декомпозировать объект на отдельные элементы и определить, какие элементы (или их взаимодействие) за какие критерии качества отвечают. Это значительно упрощает и само тестирование, и определение местонахождения ошибки.



.....

Тестировщик будет больше цениться в команде, если он будет говорить разработчику не просто: «В программе есть ошибка!», а «В программе есть ошибка *в этом* элементе». Поэтому дополнительной ответственностью тестировщиков является максимально возможное определение места ошибки *и* условий, при которых эта ошибка возникает. Деление продукта на составляющие обычно выполняется по функциональному назначению элементов, например деление на модуль взаимодействия с базой данных, модуль обработки данных, модуль пользовательского интерфейса.

.....

## 8.2 Тестовые случаи



.....

Для проверки некоторого критерия качества тестировщик должен придумать **тестовый случай** (*test-case*) — последовательность шагов, результатом выполнения которых станет определение ошибки по заданному критерию качества.

.....

На один критерий качества может быть написан не один тестовый случай, а целое множество тестовых случаев, каждый из которых некоторым образом проверяет соответствие объекта требованиям. При этом каждый тестовый случай должен отвечать на один строго заданный вопрос. Постановка вопроса зависит от того, что именно тестируемый хочет проверить данным тестовым случаем.

Каждый тестовый случай должен содержать:

- ожидаемый результат (expected result);
- фактический результат (actual);
- последовательность шагов для достижения фактического результата (steps/instructions).

Технически, суть исполнения тестового случая заключается в выполнении последовательности шагов и сравнении фактического результата с ожидаемым. Если фактический результат отличается от ожидаемого — тестовый случай считается проваленным (fail), а значит, обнаружена ошибка. Если фактический результат совпадает с ожидаемым — тестовый случай успешен (pass). Других вариантов завершения тестового случая быть не может.

Помимо трёх указанных обязательных элементов, у тестового случая могут быть описаны следующие атрибуты.

*Уникальный идентификационный номер* (Unique ID) — позволяет упростить учет и ведение тестовых случаев. В дальнейшем при взаимодействии тестируемых с разработчиками им будет достаточно сказать: «Тестовый случай №01304 провален. Нужно исправить». И разработчику будет достаточно обратиться к документации с описанием всех тестовых случаев. Это проще, чем каждый раз указывать разработчику, при какой последовательности шагов в программе возникла ошибка.

*Приоритет* (Priority) — позволяет установить важность тестовых случаев относительно друг друга. В дальнейшем, исправление ошибок, возникших на тестовых случаях с большим приоритетом, является более критичным для проекта.

*Описание/Идея* (Description/Idea) — содержит пояснение, что именно проверяет данный тестовый случай. Данное поле очень важно с точки зрения поддержки актуальности тестовых случаев и определения покрытия объекта тестами.

Тестовые случаи могут группироваться в *тестовые наборы* (test-suite). Группировка выполняется на усмотрение тестируемого исходя из собственных соображений либо согласно общей договоренности с другими тестирующими и разработчиками. Основная цель формирования тестовых наборов — упрощение навигации по тестовым случаям. Логично предположить, что группы должны определяться без возможных пересечений, т. е. один тестовый случай строго принадлежал только одному тестовому набору.

Представление тестовых случаев может быть различным, от обычного формального описания и таблиц до карточек и псевдокода. Представление тестовых случаев в виде, например, псевдокода облегчает в дальнейшем их автоматизацию. Способ хранения данного документа с перечнем всех тестовых случаев также может быть различным — на бумаге в печатном виде, на общем сервере или хранилище данных, либо системе учета ошибок. Главное, чтобы данный документ был легкодоступен и удобен в использовании.

После определения того, как объект должен себя вести, необходимо разобраться, как тестировать объект.

### 8.3 Классификация тестов

Существует много классификаций тестов по разным признакам. Тут же стоит отметить, что данные классификации пересекаются и один тест может одновременно относиться к нескольким классификациям. Например, может существовать негативный функциональный блочный тест на основе белого ящика. Использование конкретной классификации позволяет разработчикам и тестировщикам быстрее и лучше понять, какие элементы программы каким образом должны быть протестированы. Итак, тесты различаются.

По знанию внутренней системы:

- Черный ящик — тестирование объекта, основанное на знании входных и выходных параметров, но без представления о его внутреннем устройстве. В данном случае, нам не важно, какой именно механизм работы заложен в объекте — тестовые случаи строятся на предположении того, как именно этот объект будет использоваться, т. е. по предполагаемому поведению пользователя. Пример такого поведения: пользователь калькулятора — он не знает, как именно выполняется расчет квадратного корня, но он ожидает, что квадратный корень из четырех — это два; разработчик, использующий классы и методы, написанные другим разработчиком, — он не знает, как реализована индексация элементов в стороннем классе, но он будет ожидать, что индексация начнется с нуля.
- Белый ящик — тестирование объекта, основанное на знании внутреннего устройства объекта. Если мы знаем, что в алгоритме объекта есть условный оператор `if`, то мы должны выполнить такие тесты, которые проверили бы обе условные ветки алгоритма. Психологическим минусом подобного тестирования является то, что когда человек знает принцип работы, он подсознательно начинает писать заведомо успешные тесты, вместо проверки потенциально ошибочных моментов.
- Серый ящик — тестирование объекта с некоторым знанием о его внутреннем устройстве. Является симбиозом тестирования черным и белым ящиками. В данном случае тесты основываются на предполагаемом поведении пользователя, на при этом используется знание о принципе работы объекта.

По объекту тестирования:

- Функциональное тестирование — тестирование функциональности объекта, т. е. правильно ли объект выполняет свои функции. Фактически, выполняется проверка правильности выходных данных при соответствующих входных.
- Тестирование пользовательского интерфейса — проверка правильности работы пользовательского интерфейса. Например, позволяет ли интерфейс ввести некорректные данные в поля формы, такие как отрицательная зарплата работника или имя, содержащее знаки препинания.

- Тестирование локализации — проверка правильности адаптации объекта под различные языковые и культурные особенности. В простейшем случае, заключается в проверке правильности перевода элементов интерфейса. Более специфичными проверками является проверка правильности работы программы при вводе не английских символов (русских, арабских, китайских) или учет отличия разделителя целой и дробной части вещественных чисел в различных странах.
- Тестирование скорости и надежности — проверка скорости работы объекта при различных нагрузках. Например, пользователи, использующие специализированное программное обеспечение для сложных математических расчетов, предполагают, что расчеты могут длиться достаточно долго. Однако интернет-пользователи будут ожидать загрузки сайтов и контента в течение нескольких секунд. Вероятно, ваш сайт правильно выполняет все функции, описанные в техническом задании, однако если он работает очень медленно — заказчик потеряет клиентов. Поэтому помимо функционального тестирования необходимо выполнять и тестирование скорости. При этом стоит учитывать, что нагрузка на тот же сайт в разное время может быть различной — от одного пользователя до нескольких тысяч. Тестирование надежности предполагает проверку правильности работы объекта при различных вариантах нагрузок: естественная, предельная, запредельная. Что является естественной нагрузкой, а что является предельной — определяется исходя из назначения программы и вариантов её использования.
- Тестирование безопасности — поиск уязвимостей объекта, позволяющих нарушить целостность и режим доступа данных. Узкоспециализированный вид тестирования, при котором выполняется проверка того, что никто из числа пользователей или третьих лиц не может получить доступ к данным других пользователей или внутренним данным самой системы.
- Тестирование опыта пользователя — проверка пользовательского интерфейса на соответствие пользовательскому опыту и эргономике. Например, пользователь всегда будет ожидать, что главное меню программы находится в верхней части, а сочетания клавиш Ctrl+S приводят к сохранению проекта. Если ваш интерфейс будет нарушать подобные общепринятые нормы, пользователь может отказаться от использования программы.
- Тестирование совместимости — проверка правильности работы объекта в различных условиях использования. В разработке ПО такими условиями могут являться аппаратные средства, платформы/операционные системы, а также иные пользовательские программы. В данном виде тестирования вы должны определить тот диапазон условий, в которых должна работать ваша программа — от минимальных и рекомендуемых системных требований, до перечня операционных систем. Также важно учитывать, что на правильность работы вашей программы может повлиять не только ОС, но и другие программы, установленные на данной машине. Например, если вы разрабатываете некоторое сетевое приложение, вам нужно удостовериться, что ваше приложение и другие приложения пользователя не используют одни и те же порты для передачи данных.

По времени проведения тестирования:

- Альфа-тестирование — тестирование, выполняемое до передачи продукта конечному пользователю.
- Приёмочное тестирование — альфа-тестирование, выполняемое перед представителем заказчика для подтверждения работоспособности продукта по основным функциональным возможностям. После приёмочного тестирования продукт отдается представителю заказчика для более детального тестирования согласно всему техническому заданию, в то время как приёмочное тестирование демонстрирует заказчику, что продукт работоспособен.
- Регрессионное тестирование — альфа-тестирование, выполняющее проверку правильности выполнения ранее реализованной функциональности после добавления новой функциональности.
- Бета-тестирование — тестирование, выполняемое конечным пользователем. Подразделяется на закрытое и открытое бета-тестирование. В открытом бета-тестировании может поучаствовать любой конечный пользователь, которому предоставляется фактически законченный продукт. Данный вид тестирования очень популярен для интернет-ресурсов, так как тестирование выполняется в реальных условиях эксплуатации и с реальной нагрузкой пользователей. Закрытое бета-тестирование выполняется на ограниченной и малочисленной группе конечных пользователей. Закрытое бета-тестирование также максимально приближено к реальным условиям работы, однако, в отличие от открытого бета-тестирования, нагрузка на продукт или на команду поддержки продукта легко регулируется количеством бета-тестировщиков.

По ожидаемому результату:

- Позитивное тестирование — проверка на корректность работы системы при корректных входных данных.
- Негативное тестирование — проверка на реакцию системы при заведомо некорректных входных данных. Реакцией системы на некорректные данные может быть сообщение об ошибке или продолжение работы программы (в зависимости от серьезности ошибки и возможных последствий), но реакция в подобной ситуации должна быть. Если из-за необработанного негативного сценария использования реакцией системы будет аварийное завершение программы с потерей всех пользовательских данных — с большой долей вероятности пользователь откажется от вашего продукта. Поэтому негативное тестирование, как правило, более важно по сравнению с позитивным тестированием (с точки зрения отказоустойчивости), однако составление негативных сценариев может стать нетривиальной задачей для тестировщика.

По степени изолированности тестируемых компонентов:

- Юнит-тестирование (от англ. «unit» — блок, модуль) или блочное тестирование, — тестирование одного выбранного блока/компонента в изоляции от других компонентов системы. Как правило, подобными компонентами являются классы или методы.



- Интеграционное тестирование — тестирование взаимодействия определенных компонентов между собой. Стоит отметить, что программисты чаще совершают ошибки именно на уровне взаимодействия компонентов, а не на уровне внутреннего устройства компонентов.
- Системное тестирование — тестирование всей системы в целом.

По степени автоматизации:

- Ручное тестирование — выполнение тестовых случаев непосредственно тестирующим без использования инструментов по автоматизации каких-либо шагов.
- Автоматизированное тестирование — выполнение тестовых случаев с помощью средств автоматизации. Так как при разработке системы её функциональность будет увеличиваться, будет и увеличиваться объём работ по тестированию. Чтобы уменьшить количество человеческих ресурсов, требуемых для тестирования, выполняется автоматизация тестовых случаев. Для этого тестовые случаи описываются в виде скриптов, которые будут выполняться специальным ПО. По выполнению скрипта можно ознакомиться с его результатами.
- Полуавтоматизированное тестирование — тестирование, при котором часть шагов, выполняемых в тестовом случае, может выполняться автоматизировано.

По степени подготовки к тестированию:

- Тестирование по документации — тестирование согласно тестовым случаям, составленным по спецификациям к продукту.
- Исследовательское тестирование — тестирование, направленное на интуитивный поиск ошибок. Фактически, техническая документация не может описать все возможные сценарии использования продукта. Исследовательское тестирование направлено на обнаружение ошибок при незапланированном поведении пользователя.

Таким образом, приступая к тестированию программного продукта, вам необходимо:

- определить приоритеты в тестируемой функциональности и критериях качества — выполняется на основе ТЗ (если данный пункт уже не написан в ТЗ);
- определить стратегии тестирования каждого критерия качества согласно спецификации выше — вы должны выбрать, как именно будет выполняться проверка качества того или иного критерия. Вероятно, что отдельные критерии могут быть протестированы не одним, а несколькими способами;
- определить тестовые случаи для каждой стратегии тестирования — фактически, определение тестовых случаев может стать нетривиальной задачей, так как необходимо найти компромисс между качеством покрытия продукта тестами, временем на проведение тестирования и легкостью обнаружения причины ошибки по тестовому случаю;
- реализовать автоматизированные тесты, если таковые запланированы, — далеко не все виды тестов можно и нужно автоматизировать. Автомати-

зированные тесты также требуют постоянной поддержки, по поддержке некоторых автоматизированных тестов в итоге может занять больше времени, чем проведение аналогичных тестовых случаев вручную;

- определить план проведения тестирования продукта — разные тесты могут проводиться в разный момент разработки в зависимости от необходимости или времени, выделяемого на тестирование. Например, юнит-тесты логичнее выполнять как можно чаще, тогда как бета-тестирование имеет смысл проводить только после готового прототипа программы или появления значительной функциональности в приложении.

Отдельно стоит упомянуть, что разные виды тестов выполняют разные люди в команде. В частности, юнит-тесты, как правило, выполняются разработчиками и качество покрытия кода тестами определяется ведущим разработчиком. Остальные виды тестирования могут выполняться другими членами команды, от руководителя проекта и специалиста по пользовательским интерфейсам, но с обязательным участием тестировщика или специалиста по контролю качества.

## 8.4 Блочное тестирование

Юнит-тестирование (блочное тестирование, «unit-testing») — тестирование отдельного элемента изолированно от остальной системы. Относительно парадигмы объектно-ориентированного программирования системой является вся программа, а отдельным элементом — класс или его метод. Юнит-тестирование предназначено для проверки правильности работы отдельно взятого класса. Чтобы исключить из результатов тестирования влияние потенциальных ошибок других классов, тестируемый класс должен быть максимально изолирован, т. е. не использовать объекты и методы других классов. Данное требование в итоге позволяет иначе взглянуть на взаимодействие классов и выполнить рефакторинг на уменьшение связности классов.

Фактически, юнит-тестирование заключается в написании некоторого класса-обёртки, который бы создавал экземпляр тестируемого класса. В классе-обёртке создаются методы-тесты, выполняющие следующий алгоритм:

1. Создаются входные параметры — тестовые данные.
2. Подают тестовые данные на вход общедоступного метода тестируемого класса.
3. Сравнивают возвращаемое тестируемым методом значение с некоторым эталоном — заранее известным результатом, который должен получиться при правильной работе данного метода. Данный эталон определяется в ТЗ, спецификациях или другой проектной документации.

Если результат работы метода совпадает с эталоном — тест пройден. В любом другом случае тест считается провальным.

По данному алгоритму проверяется каждый общедоступный метод тестируемого класса. Защищенные или закрытые члены класса тестированию не подвергаются, чтобы не нарушать инкапсуляцию класса.

Таким образом, юнит-тесты представляют программный код, который также требует постоянной поддержки и следования определенным правилам оформления. В отличие от других видов тестирования проведение юнит-тестирования — обязанность разработчиков, а не тестировщиков.



## Контрольные вопросы по главе 8

1. В какой момент разработки необходимо выполнять тестирование?
2. Какие виды тестирования бывают, и кто из команды разработки должен их выполнять?
3. Чем отличаются тест и тестовый случай?
4. Что такое юнит-тестирование?
5. К какому виду тестирования относится ревью кода?

---

## Глава 9

# ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ ПРОЦЕССА РАЗРАБОТКИ

---

В предыдущих главах был рассмотрен процесс разработки ПО, от создания ТЗ и формирования команды до организации процесса тестирования. В данной главе будут рассмотрены информационные средства, которые позволяют значительно упростить процесс разработки программного обеспечения.

### 9.1 Система управления проектами

Для того чтобы перейти к тому, что же такое система управления проектами, необходимо определить, что же такое проект.



.....  
*Прое́кт* (от лат. *projectus* — брошенный вперёд, выступающий, выдающийся вперёд) — замысел, идея, образ, воплощённые в форму описания, обоснования, расчётов, чертежей, раскрывающих сущность замысла и возможность его практической реализации.  
.....

Если рассматривать это определение с точки зрения разработки ПО, то проектом будет являться техническое задание, IDEF-диаграммы, UML-диаграммы, планы тестирования, блок-схемы и так далее.

Однако в последнее время под проектом подразумевается нечто иное — временное предприятие, направленное на достижение определенной цели, то есть проект как процесс. Отсюда можно сформулировать понятие «управление проектами» — это любые действия, помогающие оптимально достичь цели.

В данной главе будут рассмотрены цели и задачи внедрения в процесс разработки программного обеспечения для управления проектами, а также их основные особенности.



.....  
*Программное обеспечение для управления проектами, также используется определение — система управления проектами (СУП) — комплексное программное обеспечение, включающее в себя приложения для планирования задач, составления расписания, контроля цены и управления бюджетом, распределения ресурсов, совместной работы, общения, быстрого управления, документирования и администрирования системы, которые используются совместно для управления крупными проектами.*  
.....

Изначально СУП разрабатывались как в виде desktop-приложений, так и в виде web-приложений, однако сейчас в основном используются web-системы. В настоящее время создано огромное количество СУП для самых разнообразных целей, однако самыми популярными системами при разработке ПО являются JIRA, Redmine, YouTrack.

К основным функциям СУП, используемых при разработке ПО, можно отнести:

- Bug-tracking system — система отслеживания ошибок — прикладная программа, разработанная с целью помочь разработчикам программного обеспечения учитывать и контролировать ошибки и неполадки, найденные в программах. Существует практика, когда эту систему делают доступной для всех пользователей, а не только команды разработки. Это позволяет получать более оперативную обратную связь и учитывать пожелания пользователей.
- Интеграция с системами контроля версий (см. далее).
- Система работы с документацией. В СУП обычно хранится вся созданная документация по проектам, причем она постоянно обновляется, чтобы каждому пользователю в любой момент времени была доступна актуальная информация по проекту.
- Система планирования заданий.

Именно последняя система является важнейшим модулем в СУП. В данную систему вносятся все актуальные задачи по текущему проекту. После чего они назначаются на конкретных исполнителей менеджерами либо самими исполнителями. В терминах СУП одна задача называется «тикет» (ticket). Каждый тикет содержит название задачи, ее описание, временные затраты на выполнение, статус задачи. Статусы задачи, которые доступны в системе, и возможные переходы между ними определяются менеджером проекта.

Данная система позволяет:

- отслеживать текущее состояние разработки, какие задачи уже сделаны, какие в разработке и так далее;
- отслеживать временные затраты. Обычно при завершении задачи разработчик ставит время, которое он потратил на ее выполнение. При сравнении этого времени с запланированным можно сделать некоторые выводы, например о качестве планирования либо, в некоторых случаях, об эф-

фективности разработчика. Также СУП позволяет формировать различные диаграммы, на основе которых можно сделать выводы о состоянии проекта;

- следить за загруженностью разработчиков.

Следует также отметить, что многие СУП позволяют вести управление, основываясь на одной из методологий разработки. Часто эта функциональность добавляется за счет использования плагинов либо прямо «из коробки». Например, существуют модули для Redmine, позволяющие вести управление по гибким методологиям, то есть она позволяет проводить планирование спринтов, вести учёт необходимого для реализации функционала, графики выполнения задач и т. д. Также существуют модули для реализации других методологий.

Таким образом, СУП — мощный инструмент, позволяющий проводить эффективное планирование и так же мониторинг текущего состояния проекта, что способствует более эффективному управлению проектами.

## 9.2 Системы контроля версий

При разработке программного обеспечения рано или поздно приходится вносить сложные исправления, в которых, с большой вероятностью, могут содержаться ошибки. Если проект небольшой, то, обычно, делается его резервная копия. Однако очевидно, что для больших проектов такой подход не оптимален.

Копировать каждый раз весь проект, вручную описывать версию — долго и трудоемко. Мало того, что это будет занимать слишком много места на жестком диске, так еще немудрено запутаться в версиях и использовать вместо сохраненной правильной версии промежуточную недоделанную версию с массой ошибок. Хорошо, если потом можно быстро откатиться на правильную версию, но бывают случаи, когда, наводя порядок в архиве, удаляют единственно верные копии с последними наработками.

Все это значительно усложняется, когда проект ведут несколько человек, иногда территориально удаленных друг от друга на сотни и тысячи километров. В этом случае у каждого будут образовываться свои архивы версий и возникнут значительные трудности сопровождения разработанного программного обеспечения. Также при командной разработке важно, чтобы у каждого в команде была актуальная версия исходных файлов, при подходе с ручным копированием, очевидно, возникает сложность при сборке всех изменений нескольких разработчиков в текущую рабочую копию программы.

Для решения этих проблем были созданы системы контроля версий (Version Control System, VCS) — программное обеспечение для облегчения работы с изменяющейся информацией. Рассмотрим основные функции VCS.

### Сохранение всех этапов разработки.



.....  
*Внеся изменения в один или несколько файлов проекта, программист записывает изменения в **репозиторий** — хранилище всех версий и изменений проекта.*  
 .....

Стоит отметить, что сохраняется не весь проект целиком, а, в целях экономии места и времени сохранения изменений, в репозиторий добавляются только файлы, претерпевшие изменения.

#### **Обновление до последней версии разработанного программного обеспечения.**

Так как обычно над разработкой проектов трудится целая команда специалистов, то они постоянно добавляют в репозиторий измененные файлы. Поэтому одной из основных задач системы контроля версий является возможность отслеживать все эти изменения и быстро обновлять программное обеспечение разработчиков до актуальной версии.

#### **Объединение изменений.**

Часто несколько программистов одновременно изменяют одни и те же файлы. Если изменения не пересекаются, то системы контроля версий позволяют легко и просто объединить эти изменения.

#### **Разрешение конфликтов.**

Если несколько человек изменили один и тот же участок кода, то, автоматически, объединить такие изменения невозможно. Обычно, системы контроля версий предоставляют собой инструменты, позволяющие вручную внести необходимые правки в тест программ, чтобы объединить конфликтующие части кода.

#### **Откат к предыдущим версиям.**

Если выбранное направление в развитии проекта оказалось тупиковым или содержащим ошибки, то системы контроля версий позволяют вернуть разработанное программное обеспечение к одной из последней рабочей версии, просто скопировав из репозитория нужную версию программного обеспечения либо отдельные файлы.

#### **Сопровождение нескольких направлений развития программного обеспечения.**

Не всегда можно сразу сохранять внесенные изменения. Часто приходится достаточно долго разрабатывать и отлаживать отдельные правки прежде, чем их можно объединить с основным программным обеспечением. В этом случае многие системы контроля версий позволяют организовывать параллельные ветки по контролю нескольких направлений развития программного обеспечения, быстро переключаться между ними, а затем объединять их в единое целое.

Системы контроля версий можно разделить на две группы — централизованные и распределенные.

*Централизованные системы*, такие как CVS и Subversion (SVN), для сохранения всех рабочих файлов контролируемого проекта используют репозиторий, размещенный на отдельном сервере.

В этом случае для работы с системой контроля версий каждый участник проекта сначала скачивает с сервера последнюю версию программного продукта, вносит в нее свои изменения и загружает на сервер полученный результат. При этом работа ведется с последней версией программного продукта, а если необходимо вернуться к одной из предыдущих версий разработки, что бывает достаточно часто, приходится каждый раз запрашивать сервер и скачивать необходимую версию.

После внесения всех корректировок в скаченную версию, она загружается на сервер в качестве последней версии разрабатываемого продукта. При этом придется разрешить некоторые конфликты, что бывает очень сложным, так как откат

на предыдущую версию может затронуть большое количество уже сохраненных изменений.

*В распределенных системах*, таких как Git, когда пользователи загружают данные из репозитория сервера, скачиваются все сохраненные изменения, а не только последняя версия. Естественно, каждый раз скачивать весь репозиторий не нужно, так как достаточно скопировать только те изменения, которых нет в локальном проекте пользователя. Даже несмотря на это, операции копирования данных из репозитория могут быть достаточно долгими, но это окупается при дальнейшем использовании распределенной системы контроля версий.

По сути, используя распределенные системы контроля, каждый пользователь имеет свой личный репозиторий, в который он локально сохраняет все свои изменения. При необходимости создает параллельные ветки контроля версий проекта, отслеживающие сложные изменения, которые пока что нельзя сохранять в основной версии разрабатываемого проекта.

В любой момент можно откатиться на нужную версию проекта, переключиться между ветками или объединить несколько параллельных веток в единую, содержащую все внесенные изменения. Все это делается без обращения к основному серверу.

Загрузка изменений в основной репозиторий будет производиться, когда все они отлажены и избавлены от ошибок.

В настоящее время создано множество систем контроля версий: RCS, CVS, Subversion, Aegis, Monoton, Git, Bazaar, Arch, Perforce, Mercurial, TFS. Часть из них значительно устарела (RCS, CVS), а у остальных есть свои достоинства и недостатки, и не имеет смысла проводить детальный обзор всех систем. Далее будет проведен обзор двух систем — SVN и Git. Эти системы выбраны потому, что являются наиболее распространенными и именно с ними работали авторы данного пособия.

Subversion [55] — это централизованная система управления версиями, созданная в 2004 году и основанная на технологии клиент-сервер. Является дальнейшим развитием системы CVS [56].

К преимуществам можно отнести простоту установки системы и удобство клиентов для работы с SVN. К недостаткам можно отнести зависимость от центрального репозитория и сложность (практически невозможность) работы с ветками.

Git [57] — распределённая система управления версиями файлов. Проект был создан Линусом Торвалдсом для управления разработкой ядра Linux, первая версия выпущена 7 апреля 2005 года. Система спроектирована как набор утилит командной строки, специально разработанных с учётом их использования в скриптах. Это позволяет удобно создавать специализированные системы контроля версий на базе Git или пользовательские интерфейсы.

К преимуществам можно отнести:

- высокую скорость работы;
- возможность интеграции с другими системами;
- удобный механизм работы с ветками.



Из недостатков можно отметить:

- сложность пользовательских интерфейсов клиентов Git. Достаточно сложно разобраться во всех функциях без дополнительных источников.

## 9.3 Непрерывная интеграция



.....  
*Непрерывная интеграция (Continuous Integration, CI) — практика частой сборки и тестирования проекта с целью выявления ошибок на ранней стадии.*  
.....

Непрерывная интеграция — автоматизированный процесс, в котором, как правило, используется специальное серверное ПО, отвечающее за поиск изменений в коде в системе контроля версий, сборку, развертывание и тестирование приложения.

К преимуществам использования CI можно отнести то, что разработчик в любой момент времени имеет достоверную информацию о состоянии исходных кодов в системе. В случае если последняя сборка совершилась с ошибкой, то, соответственно, не имеет смысла получать актуальные исходные файлы из репозитория, так как они не работают. Так же в этом случае происходит оповещение разработчиков об ошибке, что позволяет сразу приступить к ее поиску и устранению. Если на проекте используются юнит-тесты, их запуск при каждой сборке дает некоторую гарантию отсутствия регрессионных багов. Также можно включить в сборку различные метрики качества кода, как-то: покрытие, статический анализ, поиск дублированного кода, и т. д., автоматизировать установку на тестовую машину и тому подобные операции.

Основой системы CI является выделенный сервер с установленным на нем специальным ПО. Систем CI довольно много, вот некоторые из них — CruiseControl, CruiseControl.Net, Atlassian Bamboo, Hudson, Microsoft Team Foundation Server, TeamCity. После установки и настройки системы CI необходимо сделать конфигурации сборок для проекта. Обычно процесс сборки состоит из следующих шагов:

1. Срабатывание триггера. Цикл интеграции начинается со срабатывания триггера. Это может быть одно из следующих событий: изменение в системе контроля версий, изменение в файловой системе, определенный момент времени, сборка другого проекта, нажата «красная» кнопка, изменение на веб-сервере. Обычно настраивается несколько конфигураций с различными триггерами (например, при изменении в VCS и при наступлении ночи). Характерным примером будет случай, когда один из разработчиков вносит изменения в систему контроля версий. Для интеграционного сервера это означает, что в исходном коде проекта произошли изменения и необходимо провести сборку для проверки того, что эти изменения ничего не испортили и согласуются с ранее сделанными.
2. Обновление файлов с исходным кодом из репозитория. На данном этапе CI сервер делает обновление своей локальной копии исходного кода проекта. В процессе выясняются изменения в коде (и не только), произошед-

шие с последней интеграции. Выяснение изменений необходимо для того, чтобы в случае сбоя можно было легко выяснить причину и найти ответственного.

3. Анализ кода. После получения файлов проекта из системы контроля версий можно провести статический анализ кода. Существует множество автоматических средств, для различных языков программирования, позволяющих провести такой анализ. Обычно измеряются следующие характеристики кода: наличие типичных ошибок, статические характеристики кода (сложность, размер и т. д.), соответствие принятым стандартам кодирования и другое. Данный этап является необязательным для процесса непрерывной интеграции, но в случае его наличия можно получить дополнительные преимущества от введения практики в виде метрик по коду. Данный этап подразумевает не только получение статических характеристик кода, но и их включение в отчеты, создаваемые сервером интеграции.
4. Сборка (build). Один из основных этапов процесса это сборка проекта. Здесь происходит компиляция исходных кодов в исполнимые файлы или какой-то другой результат.
5. Выполнение модульных тестов. Модульное тестирование является неотъемлемой частью разработки приложения. Модульные тесты изначально автоматизированы, их включение в процесс интеграции крайне желательно. Поскольку часто у разработчиков нет времени или желания запускать такие тесты до того, как изменения отправлены в систему контроля версий, дополнительное их исполнение никогда не будет лишним. Дополнительную информацию можно извлечь, измеряя покрытие модульных тестов. Эта метрика поможет лучше контролировать качество выпускаемого продукта. Естественно, при отсутствии самих тестов в проекте этот этап невыполним. Хотя наличие поставленного процесса непрерывной интеграции без модульных тестов заставляет задуматься об их необходимости.
6. Защита приложения. В случае разработки desktop-приложения на данном этапе проводится обфускация (приведение исходного текста или исполняемого кода программы к виду, сохраняющему ее функциональность, но затрудняющему анализ, понимание алгоритмов работы и модификацию при декомпиляции) сборок, а также защита путем внедрением ключей и файлов лицензий.
7. Развертка или сборка инсталлятора. После того как проект собирается и все проходят модульные тесты, проект необходимо «развернуть». В случае веб-приложения это выкладывание на веб-сервер и запуск. Для desktop-приложений это сборка инсталлятора и (пере) установка в системе.
8. Регрессионное тестирование.
9. Создание архива. После того как достигнута максимальная уверенность в качестве исходного кода, необходимо сохранить его. Это можно сделать, например, посредством меток в системе контроля версий. Так же необходимо сохранить бинарные файлы проекта. Они могут понадобиться, если нужно будет воспроизвести ошибку в конкретной версии и для ручного

тестирования. CI процесс можно использовать как формализацию процесса передачи версии проекта на тестирование. К примеру можно настроить сервер публиковать свежую версию каждые две недели и сообщать об этом тестировщикам по электронной почте. Наличие регрессионных и модульных тестов является своего рода первичным тестированием и гарантирует (в некоторой степени) работоспособность данной версии. Таким образом, на тестирование не попадет версия, которая имеет существенные недостатки, препятствующие тестированию.

10. Этап генерации и публикации отчетов. Отчеты включают в себя следующее:

- причина сборки — например, изменения в репозитории;
- отчеты по статическому анализу кода;
- лог сборки;
- лог модульных тестов (какие тесты прошли и, что важнее, какие не прошли);
- лог регрессионных тестов — аналогично модульным тестам;
- статистика сборок проекта;
- все другие метрики, используемые и собираемые в проекте, — это поможет менеджеру проекта видеть все и сразу.

Среди обозначенных этапов можно выделить критические, ошибка на которых означает сбой сборки, — этап сборки, выполнения модульных тестов, выполнения регрессионных тестов. Иногда к критическим также относят этап статического анализа, например сборка считается проваленной, если оформление кода не соответствует принятому стандарту.



## Выводы

В заключение можно сказать, что CI — мощный инструмент, который позволяет автоматизировать рутинные этапы сборки, оперативно находить ошибки интеграции и следить за состоянием проекта в целом.



## Контрольные вопросы по главе 9

1. Что такое система управления проектами?
2. Какие подсистемы включает система управления проектами? Какие преимущества даёт система управления проектами?
3. Что такое система контроля версий? Какие преимущества она даёт?

4. Что такое репозиторий?
5. В чем разница между распределенными и централизованными системами?
6. Что такое непрерывная интеграция, какие этапы включает? Какие преимущества даёт непрерывная интеграция?

---

## ЗАКЛЮЧЕНИЕ

---

В заключение хочется сказать, что представленное пособие является лишь небольшой частью информации, которую должен освоить специалист в области разработки программных продуктов. Однако кажущаяся сложность области с лихвой компенсируется интересными и творческими задачами, которые приходится решать в процессе работы. Поэтому начинающим программистам не стоит бояться разработки, наоборот следует набираться опыта как можно скорее, ведь именно на определённом уровне экспертизы приходит удовольствие от занятия тем или иным делом.

Помимо перечисленного выше, хотелось бы сделать акцент на самообразовании. Ни одна специальность не даст вам всего нужного материала по разработке ПО, поэтому следующей рекомендацией будет чтение хорошей технической литературы, профессиональных блогов и программного кода именитых разработчиков. Сегодня нет проблем в том, чтобы в интернете найти открытый популярный свободно распространяемый проект и посмотреть, как нужно писать программный код и как нужно использовать те или иные архитектурные решения. Лучшим практикам удобно учиться по уже существующему опыту, худшим же, по опыту авторов, придётся учиться по большей части самостоятельно.

Хотелось бы пожелать вам успехов в освоении этой сложной и интересной области — разработки программных продуктов

---

## ЛИТЕРАТУРА

---

- [1] Dijkstra Edsger. On the Cruelty of Really Teaching Computer Science // Communications of the ACM 32. — 1989. — №12 (December). — P. 1397–1414.
- [2] Dijkstra, Edsger. The Humble Programmer // Communications of the ACM 15. — 1972. — №10 (October). — P. 859–866.
- [3] Макконнелл С. Совершенный код. Мастер-класс : пер. с англ. / С.Макконнелл. — М. : Русская редакция, 2010 — 896 с. : ил.
- [4] Gries David. The Science of Programming / David Gries. — New York : Springer-Verlag, 1981.
- [5] Knuth Donald. The Art of Computer Programming / Donald Knuth // Sorting and Searching. — MA : Addison-Wesley, 1998. — V. 3.
- [6] Humphrey Watts S. Managing the Software Process / Watts S. Humphrey. — MA : Addison-Wesley, 1989.
- [7] Plauger P. J. Programming on Purpose: Essays on Software Design / P. J. Plauger. — New York : Prentice Hall, 1993.
- [8] Beck Kent. Extreme Programming Explained: Embrace Change / Kent Beck. — MA : Addison-Wesley, 2000.
- [9] Cockburn Alistair. Agile Software Development / Alistair Cockburn. — Boston. — MA : Addison-Wesley, 2002.
- [10] Raymond E. S. The Cathedral and the Bazaar [Электронный ресурс] / E. S. Raymond. — URL : [www.catb.org/\\_esr/writings/cathedral-bazaar](http://www.catb.org/_esr/writings/cathedral-bazaar) (Дата обращения: 02.07.2014).
- [11] Heckel Paul. The Elements of Friendly Software Design / Paul Heckel. — Alameda, CA : Sybex, 1994.
- [12] Brooks Frederick P. The Mythical Man-Month: Essays on Software Engineering / Frederick P. Brooks. — MA : Addison-Wesley, 1995.

- [13] Pigoski Thomas M. Practical Software Maintenance / Thomas M. Pigoski. — New York : John Wiley & Sons, 1997.
- [14] Gilb Tom. Principles of Software Engineering Management / Tom Gilb. — England, Wokingham : Addison-Wesley, 1988.
- [15] Jones Capers. Estimating Software Costs / Capers Jones. — New York : McGraw-Hill, 1998.
- [16] Mills Harlan D. Software Productivity / Harlan D. Mills. — Boston, MA : Little, Brown, 1983.
- [17] Boehm Barry W. Improving Software Productivity // IEEE Computer. — 1987. — September. — P. 43–57.
- [18] Boehm Barry W. Industrial Software Metrics Top 10 List // IEEE Computer. — 1987. — №9. — P. 84–85.
- [19] Cooper Kenneth G. Swords and Plowshares: The Rework Cycles of Defense and Commercial Software Development Projects / Kenneth G. Cooper, Thomas W. Mullen // American Programmer. — 1993, May. — P. 41–51.
- [20] Fishman Charles. They Write the Right Stuff Fast Company // IEEE Software, 1996, December.
- [21] Haley Thomas J. Software Process Improvement at Raytheon // IEEE Software, 1996, November.
- [22] Wheeler David. Software Inspection: An Industry Best Practice / David Wheeler, Bill Brykczynski, Reginald Meeson // IEEE Computer Society. — 1996.
- [23] What We Have Learned About Fighting Defects / Forrest Shull [et al.] // Proceedings, Metrics, 2002 ; IEEE. — 2002. — P. 249–258.
- [24] McConnell Steve. Professional Software Development / Steve McConnell. — Boston, MA : Addison-Wesley, 2004.
- [25] IEEE Guide to the Software Engineering Body of Knowledge — SWEBOOK.
- [26] Фаулер М. UML. Основы. Краткое руководство по стандартному языку объектного моделирования / М. Фаулер. — СПб. : Символ-Плюс, 2011. — 192 с. : ил.
- [27] ГОСТ 19.201-78. Единая система программной документации. Техническое задание. Требования к содержанию и оформлению. — М. : Изд-во стандартов, 1980. — 5 с.
- [28] IEEE 1016-2009. Standard for Information Technology, Systems Design, Software Design Descriptions. — 2009. — 44 p.
- [29] Белбин Рэймонд Мередит. Team Roles at Work // Рэймонд Мередит Белбин. — М. : Гиппо, 2003. ISBN 5-98293-005-9, 0-7506-2675-5.

- [30] Winston W. Royce. *Managing The Development Of Large Software Systems* // IEEE WESCON Proceedings. — 1970, August. — P. 1–9.
- [31] Chaos Manifesto // A methodologies usage report [Электронный ресурс]. — URL : <http://www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf> (Дата обращения 25.04.2014).
- [32] Agile — манифест гибкой разработки программного обеспечения [Электронный ресурс]. — URL: <http://agilemanifesto.org/iso/ru/> (Дата обращения 25.04.2014).
- [33] Hirotaka T. *The New New Product Development Game* / T. Hirotaka, N. Ikujiro. — Harvard Business Review. — 1986, January. — P. 1–11.
- [34] Schwaber K. *Agile Software Development with Scrum (Series in Agile Software Development)* / K. Schwaber, M. Beedle. — Prentice Hall, 2002. ISBN 0-13-067634-9, 158 p.
- [35] Бек К. *Экстремальное программирование* / К. Бек. — СПб. : Питер, 2002. ISBN 5-94723-032-1, 224 с.
- [36] Бек К. *Экстремальное программирование: планирование* / К. Бек, М. Фаулер. — СПб. : Питер, 2003. ISBN 5-318-00111-4, 144 с.
- [37] Бек К. *Экстремальное программирование: разработка через тестирование* / К. Бек. — СПб. : Питер, 2003. — 224 с. ISBN 5-8046-0051-6.
- [38] Оно Т. *Производственная система Тойоты. Уходя от массового производства* / Т. Оно. — М. : Вершина. — 194 с.
- [39] Fishman Charles. *They Write the Right Stuff* / Charles Fishman. // Fast Company. — 1996, December.
- [40] Тидвелл Дженифер. *Разработка пользовательских интерфейсов* / Дженифер Тидвелл. — СПб. : Питер, 2008. — 416 с.
- [41] Вигерс Карл И. *Разработка требований к программному обеспечению* / Карл И. Вигерс. — М. : Русская Редакция, 2004. — 576 с.
- [42] Купер А. *Психбольница в руках пациентов* / Алан Купер. — СПб. : Символ-Плюс, 2005. — 217 с.
- [43] Draft Federal Information Processing Standards Publication 183 [Электронный ресурс]. — 1993. — 21 December. — 128 p. — URL : <http://www.idef.ru/documents/idef0.pdf>.
- [44] *Методология функционального моделирования IDEF0 : руководящий документ* [Электронный ресурс]. — М. : Госстандарт России, 2000. — 75 с. — URL : <http://www.nsu.ru/smk/files/idef.pdf> (Дата обращения: 02.07.2014).



- [45] Рыбалка Е. Н. Системотехника : учеб.-метод. пособие / Е. Н. Рыбалка. — Томск : Томский университет систем управления и радиоэлектроники, 2012. — 90 с.
- [46] Information Integration For Concurrent Engineering, IDEF3 Process Description Capture Method Report [Электронный ресурс]. — 1995. — September. — 224 p. — URL : [http://www.idef.ru/documents/Idef3\\_fn.pdf](http://www.idef.ru/documents/Idef3_fn.pdf) (Дата обращения: 02.07.2014).
- [47] Object Management Group [Электронный ресурс]. — URL: <http://omg.org> (Дата обращения: 02.07.2014).
- [48] Якобсон А. Язык UML. Руководство пользователя. Книга по Требованию / А. Якобсон, Дж. Рамбо, Г. Буч. — 2-е изд. — М. : ДМКпресс, 2007. — 494 с.
- [49] Фаулер М. UML. Основы. Краткое руководство по стандартному языку объектного моделирования / М. Фаулер. — 3-изд. — СПб. : Символ-Плюс, 2011. — 192 с.
- [50] Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч [и др.]. — 3-е изд. — М. : Вильямс, 2010. — 720 с.
- [51] Приемы объектно-ориентированного проектирования. Паттерны проектирования / Эрих Гамма [и др.]. — СПб. : Питер, 2007. ISBN 978-5-469-01136-1, 5-272-00355-1, 0-201-63361-2, 5-469-01136-4.
- [52] Рефакторинг. Улучшение существующего кода / Мартин Фаулер [и др.]. — СПб. : Символ-Плюс, 2008. ISBN 5-93286-045-6, 978-5-93286-045-8, 0-201-48567-2.
- [53] Савин Роман. Тестирование .com. или Пособие по жестокому обращению с багами в интернет-стартапах / Роман Савин. — М. : Дело, 2007. — 312 с.
- [54] Osherove R. The art of unit testing / R. Osherove. — NY. : Manning Publications Co., 2014. — 292 с.
- [55] Apache Subversion [Электронный ресурс]. — URL : <http://subversion.apache.org/> (Дата обращения: 02.07.2014).
- [56] CVS — Concurrent Versions System [Электронный ресурс]. — URL : <http://www.nongnu.org/cvs/> (Дата обращения: 02.07.2014).
- [57] git [Электронный ресурс]. — URL : <http://git-scm.com/> (Дата обращения: 02.07.2014).

---

# СПИСОК УСЛОВНЫХ ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ

---

*ЕСПД* — единая система проектной документации

*ООП* — объектно-ориентированное программирование

*ПО* — программное обеспечение

*ТЗ* — техническое задание

*УГО* — условное графическое обозначение

---

# ГЛОССАРИЙ

---

*Cleanroom* — методология разработки ПО, основанная большей частью на формальных методиках разработки и поддержки кода, позволяющая добиться минимального количества ошибок в расчёте на количество строк кода проекта.

*IDEF* (сокр. Integration Definition Methodology — Объединение Методологических Понятий). Семейство совместно используемых методов для решения задач моделирования сложных систем, позволяет отображать и анализировать модели деятельности широкого спектра сложных систем в различных разрезах.

*Scrum* — методология разработки ПО, входящая в семейство гибких методологий, позволяющая, за счёт организации процесса разработки, упростить приспособление разрабатываемого продукта к изменяющимся требованиям заказчика.

*UML* (сокр. Unified Modeling Language) — унифицированный язык моделирования, предназначенный для моделирования различных аспектов разрабатываемых программных систем.

*Антипаттерны* — негативные практики разработки и поддержания программного кода, с которыми сталкиваются по большей части начинающие программисты.

*Блок-схема* — распространённый тип схем (графических моделей), описывающих алгоритмы или процессы, в которых отдельные шаги изображаются в виде блоков различной формы, соединённых между собой линиями, указывающими направление последовательности.

*Вариант использования (use case)* — это описание множества последовательных действий (включая вариации), которые выполняются некоторым субъектом с целью получения результата, значимого для некоторого действующего лица. ВИ предполагает взаимодействие действующих лиц и системы или другого объекта.

*Водопадная методология (Waterfall)* — методология разработки ПО, характеризующаяся длительными итерациями и сложностью приспособления к изменяющимся требованиям заказчика.

*Высокоуровневое кодирование* — это реализация с помощью программного кода основной архитектуры программы — скелета программной системы.

*Гибкие методологии разработки ПО* — семейство методологий разработки ПО, характеризующиеся возможностью быстрой адаптации к изменяющимся требованиям заказчика и короткими итерациями выпуска программных продуктов.

*Детализированное кодирование* — это наращивание функционала — мышц программы.

*Диаграмма классов* — один из видов UML-диаграмм, позволяющий описать статический аспект программной системы за счёт описания классов и их взаимосвязей в системе.

*Диаграмма пакетов* — один из видов UML-диаграмм, позволяющий описать статический аспект программной системы за счёт более высокоуровневых сущностей, пакетов, и их взаимосвязей в системе.

*Диаграммы выполнения задач (burndown chart)* — диаграммы, используемые в гибких методологиях для отслеживания работы над задачами проекта.

*Диаграммы деятельности* — это один из пяти видов диаграмм, применяемых в UML для моделирования динамических аспектов систем: алгоритмов, бизнес-процессов, потока управления и пр.

*Единая система программной документации (ЕСПД)* — это набор государственных стандартов Российской Федерации, определяющих правила оформления программной документации.

*Канбан* — методология разработки ПО, основанная на концепции бережливого производства, позволяющая не перегружать процесс разработки и не иметь простаивающей. При разработке обычно используется Канбан доска.

*Команда* — это группа индивидов, которые распределяют между собой рабочие операции и ответственность за получение конкретных результатов.

*Логирование* (с англ. *log* — системный журнал) — процесс сохранения данных о работе программы с целью дальнейшего улучшения пользовательского опыта взаимодействия с программой либо исправления возникающих в процессе работы ошибок.

*Методология разработки ПО* — это учение о методах, методиках, способах и средствах разработки ПО.

*Непрерывная интеграция (Continuous Integration, CI)* — автоматизированный процесс, в котором, как правило, используется специальное серверное ПО, отвечающее за поиск изменений в коде в системе контроля версий, сборку, развертывание и тестирование приложения.

*Объектно-ориентированное программирование* — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

*Оптимизация кода* — модификация системы для улучшения её эффективности.

*Парное программирование* — практика разработки ПО, при одновременном участии пары программистов, один из которых пишет код, а второй следит за корректностью используемых решений. Через определённое время роли в паре меняются.

*Поведенческие паттерны* — один из видов шаблонов проектирования программных систем, предназначенных для распределения обязанностей между объектами в системе.

*Порождающие паттерны* — один из видов шаблонов проектирования программных систем, предназначенных для создания новых объектов в системе.

*Программирование* — процесс создания компьютерных программ.

*Проект* (от лат. *projectus* — брошенный вперёд, выступающий, выдающийся вперёд) — замысел, идея, образ, воплощённые в форму описания, обоснования, расчётов, чертежей, раскрывающих сущность замысла и возможность его практической реализации.

*Проект программной системы* — это документ, описывающий будущую программную систему, взаимодействие её частей, ответственность каждой части системы таким образом, чтобы можно было приступить к реализации программы без значительных модификаций документа.

*Прототипирование* — процесс создания прототипа программы — макета программы с целью проверки пригодности предлагаемых концепций, как интерфейсных, так и архитектурных.

*Разработки ПО через тестирование (Test Driven Development)* — практика разработки программного продукта, в которой во главу ставится написание модульных тестов, а затем рабочего кода к ним. Такая практика позволяет сразу иметь протестированный программный код.

*Репозиторий* — хранилище всех версий и изменений проекта.

*Рефакторинг кода* — (англ. *refactoring*) или реорганизация кода — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы.

*Рецензирование кода (ревью)* — одна из практик разработки ПО, заключающаяся в формальных или неформальных инспекциях кода, выполняемых одним из членов команды, лучше всего самым опытным, для устранения возможных ошибок, обучения лучшим практикам написания кода и увеличения совместного владения кодом.

*Санпорт* (от англ. *support* — поддержка) — структурная единица фирмы, занимающейся разработкой ПО, отвечающая за взаимодействие с пользователями ПО, сбор обратной связи для исправления ошибок или улучшения разрабатываемого продукта.

*Система управления проектами (СУП)* — комплексное программное обеспечение, включающее в себя приложения для планирования задач, составления расписания, контроля цены и управления бюджетом, распределения ресурсов, совместной работы, общения, быстрого управления, документирования и администрирования системы, которые используются совместно для управления крупными проектами.

*Системы контроля версий (Version Control System, VCS)* — программное обеспечение для облегчения работы с изменяющейся информацией.

*Стандарт кодирования* — принцип, говорящий о необходимости использования одного общего стандарта кодирования, который отвечает за именование локальных переменных, классов, полей, методов и пр.

*Структурные паттерны* — один из видов шаблонов проектирования программных систем, предназначенных для решения задачи компоновки системы на основе классов и объектов.

*Тестирование* — это проверка соответствия объекта желаемым критериям.

*Тестовый случай* (test-case) — последовательность шагов, результатом выполнения которых станет определение ошибки по заданному критерию качества.

*Техническое задание* — документ, используемый для инициализации разработки программной системы. От полноты и качества технического задания зачастую зависит плодотворность работы между заказчиком и командой разработчиков.

*Тикет* (слэнг от английского *ticket* — задача, электронный запрос) — задача, заведённая в системе управления проектами, нацеленная на улучшение, поддержку или исправление возникающих в проекте ошибок.

*Шаблоны поведения* — закономерности в поведении всех пользователей, используемые для создания положительного пользовательского опыта работы с интерфейсом разрабатываемой системы.

*Шаблоны (паттерны) проектирования* — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

*Фреймворк* (от англ. *framework* — каркас, программная платформа) — программная платформа, используемая для решения тех или иных задач в рамках разработки ПО, созданная на основе часто используемых решений, предназначенная для уменьшения различных вариантов реализации практически одинаковой функциональности.

*Экстремальное программирование* (XP — eXtreme Programming) — одна из гибких методологий разработки ПО с минимальными формальными ограничениями разработки и большой поддержкой социальной защищённости разработчика.

*Юнит-тестирование* (блочное тестирование, «unit-testing») — тестирование отдельного элемента изолированно от остальной системы.



Учебное издание

**Калентьев** Алексей Анатольевич  
**Гарайс** Дмитрий Викторович  
**Горяинов** Александр Евгеньевич

**НОВЫЕ ТЕХНОЛОГИИ В ПРОГРАММИРОВАНИИ**

Учебное пособие

Корректор Осипова Е. А.  
Компьютерная верстка Перминова М. Ю.

Подписано в печать 29.09.14. Формат 60x84/8.  
Усл. печ. л. 20,46. Тираж 100 экз. Заказ

---

Издано в ООО «Эль Контент»  
634029, г. Томск, ул. Кузнецова д. 11 оф. 17  
Отпечатано в Томском государственном университете  
систем управления и радиоэлектроники.  
634050, г. Томск, пр. Ленина, 40  
Тел. (3822) 533018.