А. Е. Андреев, С. И. Кирносенко

АДАПТИВНЫЕ ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А. Е. Андреев, С. И. Кирносенко

АДАПТИВНЫЕ ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Волгоград
2015

УДК 681.3.06 (075)

Рецензенты:

кафедра «Информационные системы и технологии»
ФГБОУ ВО «Волгоградский ГАУ»,
зав. кафедрой д-р техн. наук, профессор *О. В. Кочеткова*;

директор ООО «Сингулярис Лаб»
канд. техн. наук *Д. И. Крыжановский*

Печатается по решению редакционно-издательского совета
Волгоградского государственного технического университета

Андреев, А. Е.

Адаптивные технологии разработки программного обеспечения : учеб. пособие / А. Е. Андреев, С. И. Кирносенко ; ВолгГТУ – Волгоград, 2015. – 96 с.

ISBN 978–5–9948–1979–1

Рассматриваются технологические процессы разработки программного обеспечения, относящиеся к так называемым быстрым или гибким адаптивным походам (agile), включая итеративное планирование, разработку через тестирование, рефакторинг, использование паттернов проектирования.

Предназначено для студентов, обучающихся по направлениям 09.03.01 и 09.04.01 «Информатика и вычислительная техника».

ISBN 978–5–9948–1979–1

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Разработка современного программного обеспечения (ПО) требует выполнения ряда сложных технологических процессов, в которые вовлечено множество людей, выполняющих различные виды работ. Организация данного процесса — важная производственная задача, качественное решение которой позволяет значительно повысить эффективность и производительность. Поэтому рассмотрение и анализ современных подходов к разработке являются актуальными.

Под адаптивными технологиями разработки авторы понимают так называемые легкие (облегченные), быстрые, гибкие подходы, адаптирующиеся к изменению требований. Столь большое количество эпитетов относятся к единственному англоязычному термину agile, что в переводе означает «гибкий, проворный». Однако проворным обычно никакой технологический процесс не называют, а прилагательное «гибкий» скорее относится к подходу или методологии в целом, чем к технологическим процессам. Таким образом, авторы решили использовать термин «адаптивные технологии», подразумевая их адаптируемость к изменению требований.

Материалы, включенные в данное пособие, использовались авторами при проведении курсов «Технологии программирования» и «Введение в разработку программного обеспечения» для студентов направления «Информатика и вычислительная техника» в течении более чем десятка лет. Излагаемые в пособии подходы к разработке программ также использовались авторами, а также выпускниками, прослушавшими упомянутые курсы, на практике при успешной разработке ряда программных информационных систем.

1 ПОНЯТИЕ О ТЕХНОЛОГИЯХ ПРОГРАММИРОВАНИЯ, ТЕХНОЛОГИЧЕСКИХ ПРОЦЕССАХ И ПОДХОДАХ

Понятия о технологии программирования, технологических процессах и подходах с незначительными изменениями приводятся нами по материалам книги [1].

«Технология программирования изучает технологические процессы» [1] разработки программного обеспечения (ПО), порядок их прохождения (стадии), жизненный цикл разработки программы («весь период ее разработки и эксплуатации» [1]).

«Технологии удобно характеризовать в двух измерениях – вертикальном (представляющем процессы) и горизонтальном (представляющем стадии)» [1].

Технологический процесс – это упорядоченная «последовательность взаимосвязанных действий, преобразующих некоторые входные данные» [1] в требуемые выходные. «Вертикальное измерение отражает статические аспекты процессов и оперирует такими понятиями, как рабочие процессы, действия, результаты деятельности и исполнители» [1].

«Стадия - часть действий по созданию программного обеспечения, ограниченная некоторыми временными рамками и заканчивающаяся выпуском конкретного продукта, определяемого заданными на данной стадии требованиями. Стадии состоят из этапов, которые обычно имеют итерационный характер. Иногда стадии объединяют в более крупные временные рамки, называемые фазами» [1]. Таким образом, «горизонтальное измерение представляет время, отражает динамические аспекты процессов и оперирует такими понятиями, как фазы, стадии, этапы, итерации и контрольные точки» [1].

«Технологический подход определяется спецификой комбинации стадий и процессов, ориентированной на разные классы программного обеспечения и на особенности коллектива разработчиков» [1].

Классическими процессами являются [1] :

1. «возникновение и исследование идеи;
2. управление;
3. анализ требований;
4. проектирование;
5. программирование (конструирование);
6. тестирование и отладка;
7. ввод в действие;
8. эксплуатация и сопровождение;
9. завершение эксплуатации» [1].

Стандарт ISO 12207 определяет следующие процессы [1]:

«1. Основные процессы :

- приобретение;
- поставка;
- разработка;
- эксплуатация;
- сопровождение.

2. Вспомогательные процессы:

- документирование;
- управление конфигурацией;
- обеспечение качества;
- верификация;
- аттестация;
- совместная оценка;
- аудит;
- разрешение проблем.

3. Организационные процессы:

- управление;
- создание инфраструктуры;
- усовершенствование;
- обучение» [1].

Фактически конкретный вид взаимосвязей между процессами и стадиями определяется конкретным технологическим подходом.

Основные группы технологических подходов [1]:

I - Подходы со слабой формализацией (ранние технологические подходы, например «кодирование и исправление»).

II – Строгие (классические, предсказуемые, жесткие) подходы :

II.1 каскадные подходы :

- классический каскадный подход («водопад»);
- каскадно-возвратный подход;
- каскадно-итерационный подход;
- каскадный подход с перекрывающимися процессами;
- каскадный подход с подпроцессами;
- спиральная модель и др.;

II.2 каркасные подходы :

- рациональный унифицированный процесс (RUP);
- унифицированный процесс (UP);
- Microsoft Solution Framework (MSF);

II.3 генетические подходы :

- синтезирующее программирование;
- сборочное (расширяемое) программирование;
- конкретизирующее программирование;

II.4 подходы на основе формальных преобразований

- технология стерильного цеха;
- формальные генетические подходы.

III Гибкие (адаптивные, легкие) подходы

III.1 ранние технологические подходы быстрой разработки

- эволюционное прототипирование;
- итеративная разработка;
- постадийная разработка;

III.2 адаптивные подходы :

- экстремальное программирование;
- SCRUM;
- прагматичное программирование;
- адаптивная разработка и др;

III.3 подходы исследовательского программирования

- компьютерный дарвинизм;
- фрагментарное программирование и др.

Классическим технологическим подходом является так называемый каскадный подход или «водопад» (1970-1985 гг). Технологические процессы (виды деятельности) для данного подхода фактически уже упомянуты выше как «классические».

Взаимосвязь между процессами и стадиями, характерная для каскадного подхода, состоит в том, что названия стадий отождествляют с названиями классических процессов (исключая управление) – на каждой стадии выполняется свой технологический процесс, завершающийся созданием определенных формальных документов. К таким документам согласно Единой системе программной документации (ЕСПД) относят техническое задание (ТЗ), пояснительные записки к эскизному и техническому проекту (ЭП и ТП) и рабочий проект (РП), включающий текст программы, ее описание, программу и методику испытаний, эксплуатационные и другие документы [2]. В ЕСПД стадии в принципе отождествляют с видом документа, который на ней подготавливается (стадия технического задания, эскизного и технического проекта, стадия

рабочего проекта и внедрения) [2]. Специфика подхода такова, что «переход к следующему процессу осуществляется только после того, как завершена работа с текущим процессом. Возвраты к уже пройденным процессам не предусмотрены» [1].

Внешний вид модели жизненного цикла каскадного подхода – водопада с указанием формируемых документов показан на рис. 1.1

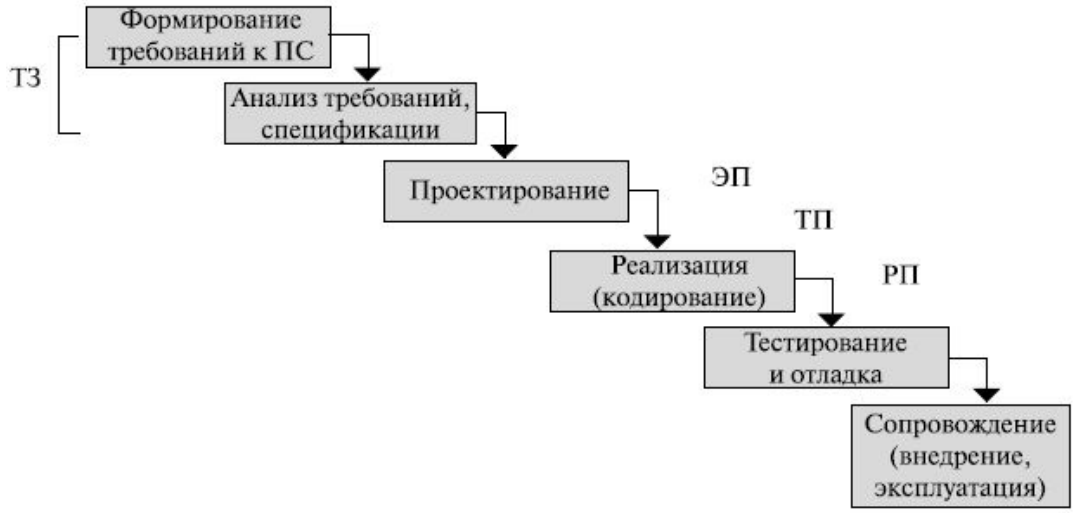


Рис. 1.1. Модель жизненного цикла каскадного подхода

Итак, водопад – это традиционный подход к разработке, в котором четко выделяются последовательные этапы. Это напоминает многоярусный водопад, при котором вода постепенно переходит от одного яруса (фазы) к другой. Каждая фаза монолитна и может продолжаться достаточно долго. Проектированию (подробному, детальному) отводится важнейшая роль. Причем проект целиком выполняется до написания кода (за исключением пробных прототипов, которые изготовляются на этапах анализа и проектирования).

Эта модель достаточно неповоротлива, и главные ее недостатки:

- отсутствие гибкости (все изменения вносятся в уже готовую программу на этапе сопровождения, либо – составляется новое техническое задание и все повторяется);

- часто трудно полностью сформулировать все требования до начала собственно программирования и до того, как заказчик познакомится с промежуточными результатами работы. Зачастую ни заказчик, ни разработчик не знают четко, чего они хотят (иногда вообще задача формулируется очень приближенно, а заказчики, тем более в наших условиях, вообще не хотят брать на себя труд участвовать в составлении и верификации требований);

- данный подход требует значительного времени.

С другой стороны, водопад достаточно предсказуем и управляем.

Недостатки классического водопада привели к появлению различных вариантов итеративных подходов на базе него (каскадно-возвратный, каскадно-итерационный, спиральная разработка и др.) В них предусматривается возврат к предыдущим этапам при необходимости, итеративность каждого этапа, а при спиральной разработке – анализ рисков и верификация на каждом из итеративных этапов.

Дальнейшее развитие итеративных подходов привело к появлению т.н. каркасного унифицированного процесса разработки (UP), в том числе в виде унифицированного процесса Rational (Rational Unified Process - RUP). Это сложный подход, включающий много концепций, в него могут встраиваться, по утверждениям авторов, различные подходы и процессы разработки. В плане жизненного цикла он отличается следующим [3]:

1) С точки зрения этапов (фаз) разработки UP в общем аналогичен «водопаду»:

- задумка (видение, начало);
- фаза развития (уточнения, исследования);
- фаза конструирования (построения);
- фаза поставки (внедрения).

Как видно, фазы в целом соответствуют модели водопада, с тем исключением, что фаза сопровождения не рассматривается как отдельная,

а составление ТЗ-ТП размыто между задумкой, фазой развития и построения.

2) Ключевой идеей подхода является его итеративность – каждая фаза, начиная с фазы развития, включает несколько итераций, на каждой из которых выполняется свой фрагмент анализа, проектирования, реализации и тестирования готового продукта (вернее его части). Задачи итерации определяются прецедентами или вариантами использования (use cases), которые отражают функциональные требования к продукту с точки зрения пользователей. Так постепенно система наращивает функциональность до полной. Кстати, оговаривается, что длительность одной итерации небольшая - от 2 до 6 недель, причем в рамках одного проекта длительность одной итерации фиксирована (от 2 для небольших до 6 для больших проектов).

3) На этапе развития принимаются ключевые решения, касающиеся архитектуры системы в целом, ее свойств, функций, используемых технологий и т.д. В конце этой фазы реализуется 10-30% системы, но все основные архитектурные решения уже приняты (до 70-80%) и на этапе конструирования в рамках этих решений система доводится до конца.

4) Большое внимание уделяется этапу задумки и анализу требований, и каждая итерация (особенно на этапе развития) предваряется серьезным проектированием, в частности, с помощью диаграмм UML (Unified modeling language – унифицированный язык моделирования, визуальный язык диаграмм [4], чаще всего используются диаграммы классов и последовательностей (кооперации)). В результате перед написанием собственно кода итерации существует фактически небольшой технический проект с указанием всех основных модулей (классов), а также – с расписанным их поведением в виде диаграмм кооперации (UML-модель, используется разработка, управляемая моделью – Model – Driven Development, MDD).

5) Требования анализируются, как и проект, на каждой итерации и в целом очень тщательно, но не до конца – действует правило 70 (80) процентов – разработчик должен представлять себе требования именно настолько, прежде чем начинать кодирование, причем в начале фазы развития только 30% требований анализируются детально.

Модель жизненного цикла UP приведена на рис. 1.2

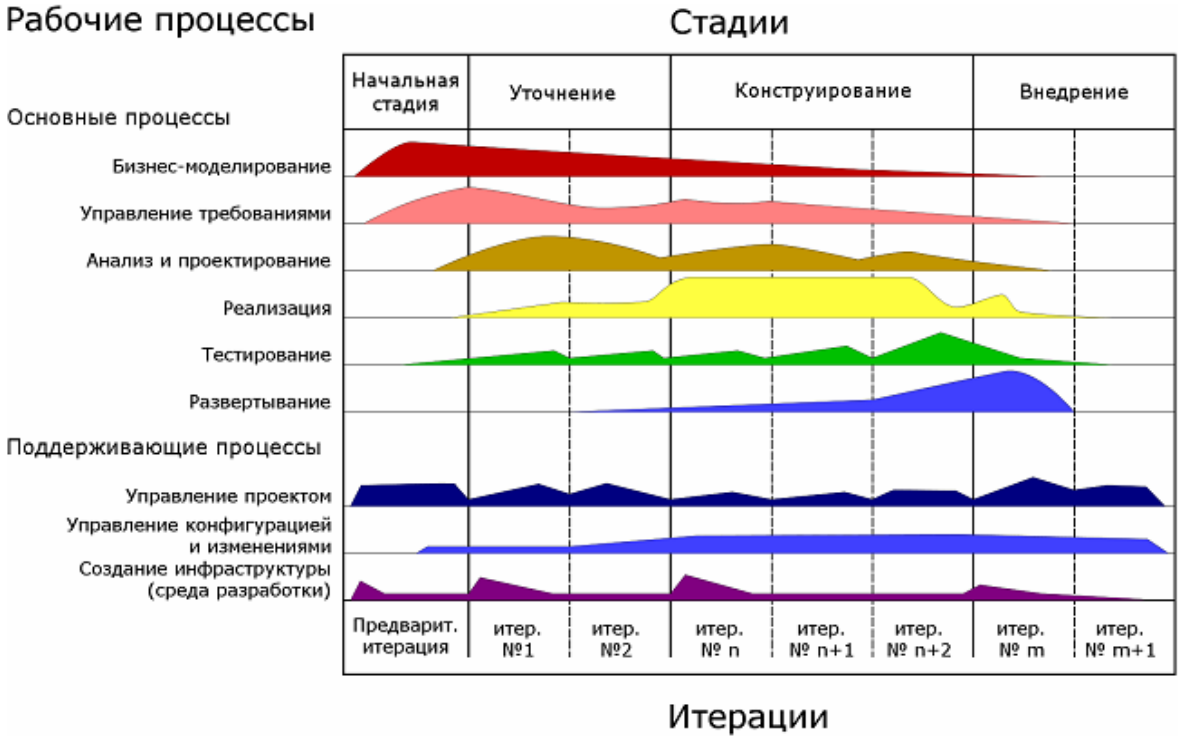


Рис. 1.2. Модель жизненного цикла UP

Как видно из рис.1.2, на большинстве итераций используются большинство видов деятельности, но идет смещение акцентов в зависимости от фазы.

Основные идеи UP изложены в многочисленных публикациях, из наиболее доступных можно назвать книгу К. Лармана [3].

Собственно RUP разработан сотрудником Rational Software Филиппом Крачтеном и вдохновлен (и поддерживается) отцами-основателями компании Rational Software, а ныне – сотрудниками IBM, авторами языка UML Г. Бучем, И.Якобсоном и Д. Рамбо (Румбахом), которых часто называют в шутку «три амигос» (три друга).

Данный подход (UP/RUP) хорош, в особенности его принцип итеративности, и универсален, но также обладает рядом недостатков:

1) Он унифицированный, то есть подходит для разнородных процессов, проектов, разработок, даже для разработок систем реального времени и аппаратных систем, что делает его немного запутанным и неконкретным (пожалуй, мало конкретных четких рекомендаций), сложным для изучения.

2) Он тяжеловат для небольших проектов и коллективов разработчиков, особенно когда бюджет и сроки проектов невелики.

3) Он требует глубокого осмысления требований, как и традиционный водопад (хотя и оставляет «на потом» 30% требований). В реальных ситуациях и 70% сразу получить и 30% детально проанализировать бывает трудно.

Особым семейством технологических подходов являются т.н. гибкие адаптивные подходы, в которых декларируется готовность разработчиков к происходящим с проектом изменениям.

Наиболее последовательно особенности адаптивных подходов были впервые раскрыты в так называемом экстремальном программировании (eXtreme Programming - XP), автором концепции которого является Кент Бек [5].

Модель жизненного цикла в XP предполагает:

- более жесткие ограничения на длительность одной итерации (от 1 до 4 недель, чаще всего - 2 недели);

- сокращение времени, затрачиваемого на анализ и проектирование (в том числе – уменьшение количества разрабатываемых диаграмм UML или вообще отказ от их разработки), уменьшение подробности анализа задач – вместо прецедентов рассматриваются короткие (в несколько предложений) «пользовательские истории» («user stories»), которые можно рассматривать как сжатый вариант прецедентов;

- правило 70% смягчается - главное быстрее приступить к написанию кода конкретной задачи на данную итерацию, все подробности уточняются у заказчика в процессе программирования;

- план и задумка всей системы составляются, но менее подробные. Общий план разработки версий составляется исходя из оценок времени на реализацию историй пользователя исполнителями с учетом приоритетов задач, определенных заказчиком. Версии состоят из итераций равной длительности (1-4 недели, чаще всего – 2). Подробный план составляется только на ближайшие 1-2 итерации. Существенное значение имеет планирование от достигнутого – по окончании каждой итерации оценивается количество решенных задач за итерацию, и план на следующую итерацию составляется, исходя из существующей скорости работы.

Эта модель призвана экономить время и средства как заказчиков, так и исполнителей. Взаимодействие с заказчиком осуществляется гораздо более плотное и дальнейшее сотрудничество зависит от результатов очередной итерации (или версии), от того, удовлетворяет ли заказчика продукт, его функции, скорость разработки, готов ли он к продолжению работы и т.д. Вместе с результатами постепенно проясняются и желания заказчика, и возможности команды разработчиков. Во главу угла ставится готовность команды к постоянным изменениям требований к программе.

В целом для небольшого коллектива и небольших проектов в условиях сжатого времени и финансирования, а также нечетко специфицированных требований, особенно в начале, этот подход, пожалуй, наиболее близок к жизни.

XP, если коротко, говорит о приоритете кодирования (конструирования) программы перед собственно проектированием, утверждая, что лучший проект – это код («The code is the design» - этот тезис излагается в статье «What is software design» Джека Ривза,

опубликованной в 1992 году и предвосхитившей появление быстрых методов разработки [6]). Также XP сравнительно легко относится к анализу требований, который проводится скорее неформально, поэтапно, эволюционно, при этом присутствует постоянная готовность к изменениям. Здесь, следует отметить, что XP не отвергает проектирование и анализ требований как таковые, так как это невозможно.

XP является гибким и простым для понимания подходом, который явился ответом на проблемы отрасли разработки ПО на рубеже 21 века, помог спасти множество застрявших проектов. Он также хорошо подходит для обучения всем итеративным процессам. Однако он не является панацеей для всех видов проектов. Разные методики по-разному относятся к доле проектирования и анализа требований в разработке. Эта пропорция должна определяться типом проекта (бизнес-приложение, ответственное приложение или встраиваемая система, от работы которой может зависеть здоровье и жизнь человека), как отмечается, например, в книге С. МакКонела “Совершенный код” [7]. Он приводит таблицу с разными типами проектов и указывает, какие процессы, приемы и компоненты процессов разработки применимы в разных типах проекта. Условно можно рекомендовать быструю адаптивную разработку для информационных систем, интернет и интранет-сайтов, бизнес-приложений; UP и RUP – для критически-важных, ответственных систем (безотказная работа в течение всего функционирования) и даже жизненно-важных систем (систем жизнеобеспечения). В последнем случае зачастую лучший вариант - применение традиционного водопада с возможностями возврата к предыдущим этапам (каскадно-возвратные подходы).

2 ИДЕОЛОГИЯ АДАПТИВНЫХ ТЕХНОЛОГИЧЕСКИХ ПОДХОДОВ

Основная идеология XP, как и других методов быстрой разработки (agile development) отражается в манифесте альянса быстрой разработки (agile manifesto). Вот как звучит этот манифест, который был подписан в конце 2001 года рядом признанных специалистов в области объектных технологий и теории проектирования [8] :

«Мы находимся в процессе поиска более эффективных методов разработки ПО, а также помогаем это делать другим пользователям. Особо пристальное внимание следует уделить таким вопросам:

- **индивиды и взаимодействия,** связанные с процессами и инструментальными средствами разработки;
- **рабочий программный продукт** и полный комплект документации;
- **совместная работа с заказчиком** и обсуждение условий контракта;
- **реакция на происходящие изменения** и соблюдение плана.

Наиболее важными являются компоненты, выделенные полужирным шрифтом, хотя не следует пренебрегать и остальными деталями.» [8]

Этот манифест был подписан Кентом Беком, Элистером Кокберном, Уордом Каннингемом, Мартином Фаулером, Робертом Мартином, Кеном Швабером и другими специалистами по разработке ПО.

Данный манифест декларирует приоритет человеческого фактора, программного кода, готовности к изменениям и общения как внутри команды, так и с заказчиком над всеми остальными компонентами проекта. Появился манифест как реакция на очередной кризис программного обеспечения, выразившийся в конце 20 века в провале многих крупных программных, прежде всего интернет-проектов («кризис доткомов»). И

хотя причины многих провалов лежат в области экономики, существенный вклад в них внесли неудовлетворительные подходы к организации разработок.

Фактически отцом – основателем, предложившим сам термин XP и его основные принципы и практики, является Кент Бек. Он обосновал идеи XP еще в середине – конце 90-х годов, но особую популярность этот подход к созданию ПО получил именно в начале 20 века. Основные методы (практики) XP приведем в изложении Кента Бека [5] :

«Методы экстремального программирования.

Игра в планирование (planning game). На основании оценок, сделанных программистами, заказчик определяет функциональные возможности и срок реализации версий системы. Программисты реализуют только те функции, которые необходимы для историй, выбранных на данной итерации. (Принцип “нам это никогда не понадобится”).

Частая смена версий (small releases). Система запускается в эксплуатацию уже через несколько месяцев после начала реализации, не дожидаясь окончательного разрешения всех поставленных проблем. Новые версии появляются часто - от ежедневного до ежемесячного выпуска.

Тесты (tests). Программисты постоянно пишут тесты для модулей (unit tests). Эти тесты собираются вместе, и все они должны работать корректно. Заказчики пишут функциональные тесты (functional tests) для историй в итерации. Все эти тесты также должны работать правильно, хотя на практике подчас приходится идти на компромисс. Чтобы принять правильное решение, необходимо понять, во сколько обойдется сдача системы с заранее известным дефектом, и сравнить это с ценой задержки на исправление дефекта.

Переработка системы (refactoring). Архитектура системы постоянно эволюционирует. Текущий проект трансформируется, при этом гарантируется правильное выполнение всех тестов.

Простой проект (simple design). В каждый момент времени разрабатываемая система выполняет все тесты и поддерживает все взаимосвязи, определяемые программистом, не имеет дубликатов кода и содержит наименьшее возможное количество классов и методов. Это правило кратко можно выразить так: «Каждую мысль формулируй один и только один раз».

Программирование в паре (pair programming). Весь код проекта пишется двумя людьми, которые используют одну настольную систему.

Непрерывная интеграция (continuous integration). Новый код интегрируется в существующую систему не позднее чем через несколько часов. После этого система вновь собирается в единое целое и прогоняются все тесты. Если хотя бы один из них не выполняется корректно, внесенные изменения отменяются.

Метафора (metaphor). Общий вид системы определяется при помощи метафоры или набора метафор, над которыми совместно работают заказчик и программисты.

Коллективное владение (collective ownership). Каждый программист имеет возможность в любое время усовершенствовать любую часть кода в системе, если он сочтет это необходимым.

Заказчик с постоянным участием (on-site customer). Заказчик, который все время работы над системой проводит вместе с командой разработчиков.

40-часовая неделя (40-hour weeks). Объем сверхурочных работ не может превышать по длительности одной рабочей недели. Даже отдельные случаи сверхурочных работ, повторяющиеся слишком часто, являются

сигналом серьезных проблем, которые требуют безотлагательного разрешения.

Открытое рабочее пространство (open workspace). Команда разработчиков располагается в большом помещении, окруженном комнатами меньшей площади. В центре рабочего пространства устанавливаются компьютеры, на которых работают пары программистов.

Не более чем правила (just rules). Если вы входите в коллектив, работающий по технологии XP, вы обязуетесь выполнять изложенные правила. Однако это не более чем правила. Команда может в любой момент поменять их, если ее члены достигли принципиального соглашения по поводу внесенных изменений.» [5]

Роберт Мартин [8] добавляет еще ряд принципов:

«Стандарты кодирования - программный код выглядит единообразно, соответствуя самым высоким требованиям к качеству.

Постоянный темп - команда разработчиков работает длительный срок. Их тяжелая работа должна выполняться равномерно, без «марафонских рывков» …» (Последнее требование соответствует требованию об отказе от сверхурочной работы).

От себя добавим, что источники по XP постоянно советуют также:

- использовать для повышения гибкости и улучшения кода паттерны проектирования;

- не усложнять систему (даже с целью увеличения гибкости) до того, как это действительно будет необходимым (снова принцип «Это вам никогда не понадобится» - «You Aren’t Gonna Need It» - YAGNI);

- для ООП-разработчиков советуют следовать базовым правилам объектного проектирования [8], для всех – просто разумным правилам сохранения простоты и понятности кода (не допускать дублирования кода, все делать максимально простым образом, использовать выразительные комментарии и имена переменных и функций и так далее).

Многие авторы прямо указывают, что принципы и практики XP следует вводить постепенно. С нашей точки зрения, помимо собственно жизненного цикла проекта из XP можно непосредственно сразу (или очень быстро) начать применять следующие практики:

- модульные тесты (unit tests);
- переработку кода (рефакторинг);
- парное программирование;
- простой проект (хотя бы принцип устранения дублирования кода);
- инкрементное динамическое планирование.

Несколько сложнее начать применять паттерны, но именно они в сочетании с тестами, рефакторингом, итеративным планированием и разработкой позволяют адаптироваться к изменению требований. Поэтому в следующем разделе мы рассмотрим именно перечисленные практики (технологические процессы), вынеся за скобку парное программирование, которое можно освоить только на практике.

3 ОБЩИЕ ТЕХНОЛОГИЧЕСКИЕ ПРОЦЕССЫ АДАПТИВНОЙ РАЗРАБОТКИ

3.1 Планирование

Экстремальное программирование стоит, если так можно выразиться, на 3 китах: планирование, тестирование и рефакторинг. Как же выглядит экстремальное планирование ? Подробно о нем рассказано в книге К.Бека и М.Фаулера, которая так и называется: «Экстремальное программирование: планирование» [9]. Естественно, что мы не можем в небольшом пункте пособия рассмотреть содержание всей книги, но сформулируем основные особенности планирования в XP.

Планирование выполняется в адаптивных подходах :

- для отбора задач для текущей итерации;

- для реакции на изменения и неудачи;

- для того, чтобы не взять на себя много;

- для того, чтобы не брать на себя мало;

- для того, чтобы знать, как идут дела.

Если сформулировать в трех пунктах, мы планируем :

- чтобы заниматься самыми важными делами на данный момент;

- чтобы согласовывать свои действия с действиями других людей (с другими разработчиками при командной разработке, с заказчиком – всегда);

- чтобы знать, что делать при возникновении нештатных ситуаций при выполнении двух предыдущих пунктов.

Итак, считаем, что мы провели начальный этап анализа разработки, возможно, подумали об архитектуре нашего приложения (собственно, только эскизно). Да, еще, возможно, мы написали несколько прототипов (небольших программ, которые можно создать за 1-2 дня, а лучше меньше). Что мы делаем дальше? Мы должны заняться планированием. Но не стоит думать, что планирование в XP это какой-то этап разработки – это, скорее, процесс, как и проектирование, и тестирование, и рефакторинг. Планирование должно выполняться постоянно.

В целом можно выделить ряд ключевых особенностей экстремального планирования:

1) Итеративность разработки и плана.

2) Планирование выполняется постоянно. План постоянно подвержен изменениям.

3) Составляются разные по масштабу, степени подробности и временному интервалу планы : план версий (в масштабе идеальных недель, состоящих из 5 идеальных дней), план текущей итераций (в идеальных днях), план задач на день и т.д.

4) Планы версий и итераций более всего корректируется на границах итераций и версий, хотя могут корректироваться и в процессе итераций. Само планирование выполняется ежедневно.

5) Одним из ключевых моментов планирования является измерение производительности команды разработчиков. Производительность измеряется в задачах, решенных за «идеальный день», соответственно план расписан в идеальных днях (идеальный день – это 8 часов, полностью посвященных решению задачи с нормальной работоспособностью).

6) Для оценки объема работ и производительности используют оценки аналогичных работ, выполненных в прошлом. В ходе проекта объемы и скорость пересчитывают исходя из текущей скорости – планирование ведется от достигнутого.

7) Одной из ключевых идей планирования (и всей книги [9]) является следующая: если разработчики не успевают выполнить определенный объем работ, то это означает, скорее всего, что он просто слишком велик, и его нужно пересмотреть в сторону уменьшения. Это выполняется, естественно, заказчиком. Разработчики не говорят: «Нам не хватает времени», они говорят: «У нас слишком много дел». Речь не идет об уловках – речь идет только о трезвой оценке своих возможностей и борьбе со стрессом и переработкой.

8) Активное участие в планировании (как и во всей разработке) принимает заказчик. Он принимает решение о том, какие пользовательские истории включать в какие итерации и версии, что является более критичным и важным, чем можно пожертвовать.

9) Список задач для выполнения управляется также рисками, то есть теми аспектами, которые более всего влияют на степень неопределенности и тем самым на ход проекта. Приоритеты определяются с учетом рисков и пожеланий разработчиков. Если есть две задачи,

равнозначные по приоритету для заказчика, разработчики выбирают для реализации ту задачу, которая может ответить на большее число вопросов и снизить риск разработки.

3.2 Модульное тестирование

3.2.1. Роль тестирования в XP. Приемочные и модульные тесты. Пример модульного теста. Критика модульных тестов.

Тестирование, как уже упоминалось, представляет собой второй (но не по важности) столп, одного из трех китов, на которых базируется XP. Не случайно ему посвящена отдельная книга Кента Бека [10]. Тестирование в XP носит исчерпывающий, всеобъемлющий характер. К тестированию относятся очень серьезно и строго. Роль тестирования не сводится к поиску и исправлению ошибок, для чего традиционно выполняются тесты в других подходах.

В XP выделяют два основных видов тестов: 1) приемочные или функциональные тесты и 2) модульные тесты.

Приемочные тесты (acceptance tests) разрабатываются в идеале заказчиком, либо тестировщиками по заданию заказчика и служат для приемки отдельного этапа работ или проекта в целом. Они разрабатываются на уровне прецедентов. Лучше, когда приемочные тесты автоматизированы, но в общем случае это просто набор входных данных для прецедента, описание процедуры их ввода в систему, последовательности действий с системой и ожидаемого результата.

Модульные тесты (unit tests) разрабатываются самим разработчиком в виде тестов для всех небольших модулей, из которых состоит приложение (уровня одного класса).

В качестве стартового примера модульного теста рассмотрим фрагмент примера тестирования класса даты Date из книги Брюса Эккеля Thinking in C++ (в русском издании «Философия C++») [11].

Сама функция для выполнения тестовых проверок может выглядеть так :

```
int nPass = 0, nFail = 0;
void test(bool t) { if(t) nPass++; else nFail++; }
```

Начальный модульный тест на корректность создания объекта с помощью конструктора и отработку его основные методов доступа к полям и метода преобразования в строку :

```
int main() {
        Date mybday(1951, 10, 1);
        test(mybday.getYear() == 1951);
        test(mybday.getMonth() == 10);
        test(mybday.getDay() == 1);
        test(mybday.toString() == "19511001");
        cout << "Passed: " << nPass << ", Failed: "  << nFail << endl;
}
/* Ожидается вывод: «Passed: 4, Failed: 0» */
```

Модульные тесты преследуют несколько иные цели по сравнению с приемочными.

Основные цели модульного тестирования в XP:

1) Поиск и исправление ошибок (средство отладки).

2) Приемка готового продукта (средство приемки).

3) Верификация проекта (оценка устойчивости и целостности разработки).

4) Средство поддержки улучшения проекта (рефакторинга).

5) Средство документирования модулей (как их использовать).

6) Средство разработки (в разработке, управляемой тестами).

Так как проектирование, рефакторинг и внесение изменений происходят постоянно (это процессы, а не этапы), то и тестирование – это перманентный процесс, который не выполняется на отдельном этапе отладки проекта, а выполняется всегда. Всеобъемлющий характер модульного тестирования означает, что ни один модуль не считается разработанным, если к нему не прилагается набор тестов, полностью (насколько это можно оценить) тестирующих его заявленную функциональность. Фраза "я уже написал программу, осталось только ее отладить" в XP не имеет смысла. Работа над модулем завершается (на данном этапе, итерации), когда отрабатывают все тесты.

Подход к поиску ошибок: мы не ждем ошибки, а провоцируем их, строим тесты для всей функциональности, а набор тестов постоянно расширяется и остается в рабочем состоянии.

И самое необычное - при использовании подхода разработки через тесты (Test Driven Development - TDD) тесты являются инструментом разработки.

Вкратце это работает так: придумывая тесты, вы придумываете интерфейсы и проектные решения для вашей задачи. Главное здесь - начинать писать тесты модулей до того, как начинается написание кода самих модулей (принцип «Вначале Тесты» – «Tests First»).

Если обобщить возражения критиков модульного тестирования, можно привести следующие основные критические утверждения [12] :

1) Исчерпывающее тестирование сразу приводит к мысли о двойном объеме работ. (Кстати, как и парное программирование – другая практика XP). То есть, вместо одного проекта приходится писать два. Да, это так. Казалось бы, это представляет основную проблему. И это действительно основной аргумент противников модульного тестирования в XP.

На самом деле только так можно (по мнению идеологов XP) добиться устойчивости проекта, над которым работают разные люди, в разное время, который состоит из большого числа взаимодействующих модулей и постоянно меняется. Тестирование неизбежно при постоянном изменении проекта, как с точки зрения его функций, так и с точки зрения проектных решений и кода.

Что происходит при экономии времени на тестирование ? На шее разработчика затягивается зловещая петля "Не хватает времени на тестирование" [10]. В условиях ограниченного времени, большого объема работ и отсутствия систематически проводимых, всеобъемлющих и готовых к работе наборов тестов по мере усложнения проекта разработчик вынужден тратить все больше времени на несистематический, хаотичный поиск ошибок, который малоэффективен и приводит к усугублению ситуации.

Если ваш опыт говорит о том, что тесты не нужны, значит вам не приходилось работать в таких условиях, либо объем и сложность задачи были недостаточными, либо вы выходили из ситуации с большими затратами времени и сил.

В любом случае тестировать все равно придется, но только модульное тестирование позволяет автоматизировать процесс многократного повторного тестирования, которое неизбежно при внесении любых изменений в код. Не жалейте времени на тестирование – оно окупится.

2) Тесты мешают думать над основной задачей.

«Некоторые разработчики чувствуют, что инкрементальное юнит тестирование прерывает процесс. Они пишут большой кусок сложного кода для приложения, и беспокоятся, что остановка для того, чтобы написать тест приведет к тому, что они забудут, на чем остановились. Это ошибочная концепция потому, что как только вы начнете писать юнит

тесты, вы сразу увидите, что тесты очень часто определяют приложение. Ваша креативная работа выливается в придумывание и написание тестов. Кодирование самого приложения превращается в серию управляемых шагов, последовательному добавлению кусочков кода для того, чтобы ваши новые тесты проходили. Не надо держать в голове сверхконцепцию во время программирования» [12]. Хорошую иллюстрацию к этому тезису можно встретить в книге [8].

3) Тестирование не нужно в простых ситуациях.

«Программистам нужна определенная степень уверенности - каждый день они сталкиваются с задачей создания чего-то из ничего. Однако, зачастую уверенность превращается в самонадеянность - большинство кода нормально работает, а если и нет, то это быстро исправить. Очевидно, что эта функция из четырех строк правильная, так зачем тратить время и тестировать ее? В проекте, где занят один человек, такая философия часто работает. Но даже самые лучшие разработчики делают ошибки. Фактически, лучшие разработчики, скорее всего, несут ответственность за самые хитрые ошибки. Как только такие ошибки попадают в конечное приложение, они имеют возможность взаимодействовать с другими ошибками, делая диагностику очень сложной» [12].

4) «Тестировать слишком сложно. Есть кодировщики, которые не пишут юнит тесты потому, что они уже на пределе. Они чувствуют, что добавление тестирования к имеющейся нагрузке совсем раздавит их. Именно этим разработчикам юнит тесты помогут больше всего» [12]. Тестам несложно учиться, главное понять и попробовать концепцию. Кроме того, существуют каркасы модульного тестирования, которые облегчают создание модульных тестов. Также существуют и библиотеки подставных объектов (mock objects).

5) «У меня уже есть очень много строк старого кода. Никто и не отрицает. Это очень трудно - добавить юнит тесты к старому коду. Этот

код, возможно, не структурирован так, чтобы можно было легко тестировать. Даже если и структурирован, это огромные издержки - добавить все тесты. Хорошо протестированная система будет иметь больше строк, чем рабочий код» [12].

«Наши рекомендации для такого случая прагматичны. Неразумно ожидать от разработчика работы по созданию полного набора тестов старого кода, поэтому не надо и пытаться. Вместо этого ищите наибольшую отдачу. Обычно это происходит, когда вам надо сделать изменения в старом коде. По мере изменений, пишите тесты. Если изменение вызвано исправлением ошибки, напишите сначала тест, который воспроизводит ошибку, затем сделайте так, чтобы тест проходил. Поскольку ошибки обычно водятся группами, посмотрите, не можете ли вы проработать цепочку обстоятельств, приведшую к изначальному появлению ошибки. Если вы работаете над кодом, который общается со старой системой через внешний интерфейс, напишите тесты, которые убеждаются, что интерфейс работает так, как от него ожидается. Не выбрасывайте тесты - найдите способ сохранить их вместе со старым кодом. Таким образом, ваш набор тестов будет расти со временем» [12].

6) Бывает код, который нельзя протестировать.

Это одно из оправданий, которое имеет основу. Сложно тестировать работу с оборудованием, сторонним кодом, работу с базой данных или внешними файлами, которые сами могут содержать ошибки.

«В таких случаях опять мы предлагаем принять прагматический подход. Проектируйте ПО так, чтобы можно было протестировать как можно больше кода без учета внешних интерфейсов. (Такое разделение, в любом случае, хорошая практика программирования.) Затем создайте хороший набор интеграционных тестов, чтобы создать эквивалент юнит тестирования оставшегося кода» [12].

Интерфейс пользователя (UI) - это особый случай. «Существуют технологии, позволяющие выполнить юнит тест прямо на уровне экрана, клавиатуры и мыши, но они обычно неудобны. Мы советуем разрабатывать тонкий уровень UI с хорошо определенными интерфейсами к остальной части приложения. Тестируйте юнит тестами вплоть до этого уровня и оставьте GUI для тестировщиков» [12].

3.2.2 Каркасы модульного тестирования xUnit

На практике для модульного тестирования вместо собственных тестовых функций или самостоятельно созданных библиотек обычно используют семейство каркасов тестирования xUnit, которые реализованы для многих существующих языков программирования, платформ и средств разработки. Самые известные, это SUnit для SmallTalk, написанный К.Беком в 1998 г., JUnit для Java (авторы К.Бек и Э.Гамма), cppUnit для C++, NUnit для .NET, но также можно упомянуть phpUnit, DUnit (для Delphi), PyUnit (Python), QUnit (для JavaScript) и даже SQLUnit. Все эти каркасы (за исключением SQLUnit) более-менее однотипны и следуют концепции исторически первых SUnit и JUnit.

Рассмотрим, например, open-source каркас NUnit для проектов, реализованных на платформе .NET [13]. Оболочка (и библиотека классов) NUnit предполагает использование разработчиком классов - наборов тестовых методов (Tests). Пользователь создает свои тестовые наборы, используя специальные атрибуты .NET для классов и методов. В частности, [TextFixture] – для класса – набора тестов, [Test] – для публичного тестового метода, входящего в набор, [SetUp] – метод класса для инициализации, выполняемой перед каждым тестом, [TearDown] - метод класса для очистки данных, выполняемой после каждого теста и так

далее. В качестве методов для выполнения тестовых проверок используются методы библиотечного класса Assert оболочки NUnit. Эти методы вызывают остановку выполнения тестового метода (с атрибутом [Test]) с выдачей информации о месте не невыполненного утверждения в программе и реальных значениях проверяемого параметра, если указанное в качестве параметра методов условие не выполняется. Чаще всего используется метод Assert.AreEqual (<Ожидаемое значение>, <Проверяемое значение>, [Дельта]), для сравнения двух величин с возможным указанием допустимой величины расхождений. При несовпадении величин с учетом указанного расхождения метод останавливается, печатаются ожидаемое и реальное значения, номер строки в программе. Реже используются методы Assert.IsTrue(<bool>) и Assert.IsFalse(<bool>).

Для выполнения тестирования необходимо указать оболочке файл со сборкой .NET. Оболочка по метаданным .NET и атрибутам тестовых классов и методов составляет список тестов (точнее – список списков) для данной сборки .NET (если тесты скомпилированы с основным проектом). При запуске набора тестов оболочка отслеживает их выполнение, сбои, выдает список и информацию о сбоях, общее количество удачных и неудачных тестов, время тестирования, выдает предусмотренную разработчиком отладочную печать и так далее.

Пользователю предоставляются два варианта пользовательского интерфейса оболочки: графический и консольный (а также Add-In для Visual Studio). При успешном прохождении всех тестов в графической оболочке выдается зеленая полоса, а при сбое одного или более тестов – красная полоса.

Для использования каркаса NUnit в своем проекте необходимо :

1. Установить каркас на компьютер.

2. Добавить ссылку на головную сборку оболочки (nunit.framework) в свой проект с помощью пункта контекстного меню проекта Add Reference.

3. Добавить объявление используемого пространства имен тестовой библиотеки в каждый файл с описанием класса – тестового списка:

```
using Nunit.Framework; .
```

В частности, приведенный ранее пример модульного теста для класса даты с использованием NUnit мог бы выглядеть так (в данном случае пример уже приведен на языке C#) :

```
using Nunit.Framework; .
[TestFixture]
public class TestDate {
        [Test]
        public void testCreateAndRead() {
                Date mybday(1951, 10, 1);
                Assert.AreEqual(1951, mybday.getYear());
                Assert.AreEqual(10, mybday.getMonth());
                Assert.AreEqual(1, mybday.getDay());
                Assert.AreEqual("19511001", mybday.toString ());
        }
}
```

В данном случае мы сохранили интерфейс тестируемого класса Date в стиле C++, хотя на C#, скорее всего, мы бы использовали свойства.

В современных версиях интегрированной среды разработки Microsoft Visual Studio практически во все редакции, в том числе свободно распространяемые, включен свой каркас тестирования. Для .NET проектов он основан на NUnit, хотя использует свои собственные атрибуты для указания на список тестов и на каждый тест (соответственно, [TestClass] и [TestMethod]), а для проектов на чистом C++ (не managed C++) – на основе cppUnit.

3.2.3 Примеры простых шаблонов модульного тестирования

Для облегчения модульного тестирования помимо использования каркасов для модульного тестирования следует использовать описанные в литературе приемы или шаблоны тестирования [10]. Приведем примеры некоторых простых типичных шаблонов модульного тестирования.

Шаблон Подделка (Fake It) - в тестируемом методе выполняется возврат ожидаемой в тесте константы, что должно просто обеспечивать срабатывание теста. Это преследует две цели. Основная – срабатывание теста и переход к другим шаблонам (например, к триангуляции), чтобы спокойно двигаться по списку реализованных тестов. Дополнительно подделка может выявлять достаточно необычные ошибки. Например, если после подделки тест не срабатывает, это может говорить о неправильном преобразовании типов данных при возврате результата из функции, или даже о том, что мы тестировали не тот класс, что является следствием невнимательного использования наследования или полиморфизма.

Шаблон Триангуляция предполагает использование двух тестовых проверок, которые совместно не проходят и требуют выполнения обобщения в коде. Этот шаблон поддерживает концепцию разработки через тестирование.

Шаблон Подставной объект (Mock Object) – используется для изоляции тестируемого модуля от других модулей, с которыми он взаимодействует (например, изолируется класс, работающий как источник данных). Является одним из основных приемов в модульном тестировании. Для облегчения создания подставных объектов используются библиотеки таких mock объектов.

Шаблон Строка тестирования (отчета) используется для отслеживания внутренних событий внутри модуля или сообщений, которые посылают друг другу модули при тестировании их совместного

поведения. Является вариацией отладки с помощью отладочной печати или создания файлов журналов (log), с той разницей, что вместо файла, консоли или окна сообщений информация выводится в строку, доступную для кода модульного теста. Содержимое этой строки сравнивается с ожидаемым.

3.2.4 Каталог шаблонов тестирования и рекомендации по составлению списков тестов

Каталог шаблонов тестирования [10] разделен на подгруппы.

В частности, выделяют общие шаблоны тестирования, шаблоны зеленой полосы (для быстрого ее получения), шаблоны красной полосы, шаблоны разработки через тестирование (TDD).

Общие шаблоны тестирования :

- Child Test (мелкие шаги – разбиваем тестирование сложной задачи на более мелкие шаги);
- Mock Object (поддельный объект для изоляции тестируемого);
- Self Shunt (тестовый метод вместо объекта, с которым взаимодействует тестируемый объект);
- LogString (строка журнала – описан ранее);
- BrokenTest (если вы программируете один, лучше оставлять последний тест в конце дня не работающим);
- Clean Check-In (если вы программируете в группе – все тесты в конце дня, наоборот, должны работать).

Шаблоны тестирования зеленой полосы (направлены на то, чтобы быстрее ее получить): Fake It (Подделка), Triangulate (Триангуляция), Obvious Implementation (Очевидная реализация), One to Many (От одного ко многим) [10].

Шаблоны тестов красной полосы (чтобы ее получить – то есть когда какие тесты добавлять, с другой стороны - чтобы быстрее выйти из красной полосы):

- One Step Test (тест одного шага – каждый тест для одного шага в направлении цели);

- Starter Test (первый тест небольшой, чтобы с чего-то начать, но все-таки он чему-то учит, иногда это тест приложения в целом);

- Explanation Test (приводите примеры в форме тестов при общении с другими разработчиками);

- Learning Test (изучение внешних и библиотечных модулей);

- Another Test (еще один тест – новые идеи не обсуждаются долго, а просто добавляются в тест);

- Regression Test (регрессионный тест – если встретилась ошибка, добавить тест для ее выявления – самый маленький тест, который ее выявляет, а потом заставить его работать).

Шаблоны разработки через тестирование (TDD) :

- Test (тесты нужны для всего, они должны быть автоматическими);

- Isolated Test (тесты должны быть изолированными друг от друга, изолируются также и тестируемые модули);

- Test List (планирование идет через работу над списком тестов);

- Test First (вначале тест);

- Assert First (тест начинается с assert' а, то есть с проверки);

- Test Data (тщательно готовьте данные для тестов, делайте их понятными и простыми для понимания);

- Evident Data (очевидные, понятные данные – данные могут указывать на способ решения, например, вместо сравнения с константой можно сравнивать с выражением, составленным из констант).

Кроме того, авторы книги [14] дают также дополнительные рекомендации относительно выбора тестов, известные как Right-BICEP:

Right – выбирайте правильные данные для тестирования (соответствует шаблону TestData);

Boundary – необходимо проверять граничные условия (0,1, минимум, максимум и так далее);

Inverse – для проверки можно использовать обратные функции (например, подстановка корней в уравнение и пр.);

Cross-check – проверка с помощью альтернативных алгоритмов (например, быструю сортировку через заведомо верную, но медленную);

Error – нужно вызывать ошибки и проверять реакцию на них;

Perfomance – рекомендуется оценивать в тестах производительность решения или хотя бы время отклика.

3.3 Переработка кода

Рефакторинг (переработка) – улучшение кода без изменения его функциональности (по крайней мере – наблюдаемой функциональности, потенциал кода после рефакторинга растет). Термин «рефакторинг» ввел в практику программирования Мартин Фаулер в своей книге «Рефакторинг или улучшение существующего кода» [15]. По Фаулеру, рефакторинг это «изменения, производимые во внутренней структуре программного обеспечения, которые делают его более понятным для понимания и облегчают его модификацию (снижают ее стоимость), но при этом не затрагивают его наблюдаемую функциональность (поведение)» [15].

Рефакторинг является универсальной практикой, не относящейся только к XP (как в принципе, и планирование, и тестирование). Рефакторинг можно применять в разных проектах, причем не только в объектных разработках, в разных технологических подходах разработки.

Если коротко, цели рефакторинга таковы:

1. Устранение дублирования кода (последствия Copy-Paste).

2. Облегчение модификации кода.

3. Сделать код более понятным (легко читаемым).

Говоря о рефакторинге, часто используют метафоры «кода с душком» и «уборки на кухне». Код с душком, это «плохой» код, или код, который постепенно становится плохим. Плохой код коррелирует с плохим проектом. В данном случае речь идет о микроуровне проекта, а не о макроуровне, как в случае архитектуры. Метафора уборки на кухне говорит о том, что если плохой код – это остатки продуктов, мусор, грязь на тарелках и прочие проблемы, возникающие на кухне, которую долго не убирали, то рефакторинг – это процесс наведения порядка на этой самой кухне. Причем, что очень важно, эта метафора говорит нам о том, что лучше всего проводить уборку постоянно, маленькими порциями, а не накапливать проблемы. То есть поел – убрал, помыл посуду. Набрался мусор – вынес его. В противном случае однажды, придя на кухню, во-первых, не сможешь поесть из-за отсутствия чистой посуды, во-вторых, неприятный запах распространится на всю квартиру, неудобно принимать гостей и так далее. То же примерно происходит и с кодом. Завершение трапезы – это срабатывание теста или набора тестов. Уборка – это рефакторинг. Его можно долго не делать, но потом это скажется.

3.3.1. Некоторые признаки плохого кода («с душком»)

Дублирование кода (Duplicated Code) – очень часто встречающаяся ситуация и одна из самых проблемных: два и более одинаковых (или похожих) фрагмента как минимум приводят к необходимости синхронизировать все изменения в разных фрагментах, размер кода увеличивается, он становится трудно обозримым и пр.

Длинный метод (Long Method) – метод процедурного типа, который содержит много кода и много последовательных действий. Типичный признак процедурного, а не объектного стиля программирования. Такой

метод рекомендуется разбить на несколько, а на возможные границы методов часто указывают комментарии.

Комментарии (Comments) – сами по себе комментарии не являются проблемой, но часто они либо а) приводятся для того, чтобы прокомментировать запутанный фрагмент кода, либо б) приводятся, чтобы прокомментировать очередной шаг в длинном методе. Оба эти случая нуждаются в рефакторинге.

Операторы типа switch (Switch Statements) - наличие операторов выбора с большим числом вариантов (как и вложенные ветвления) также характерно для процедурного стиля, они приводят к дублированию кода и к увеличению длины методов, а также затрудняют понимание кода и внесение изменений.

Длинный список параметров (Long Parameter List) – слишком большой список параметров, усложняющий вызов функции / метода. Сами по себе параметры не вредны, однако нужно стараться, чтобы их не было слишком много (ну, скажем, более 10). Части параметров можно присваивать значения по умолчанию, в ряде случаев можно вместо вписка параметров использовать параметр-объект.

Расходящиеся модификации (Divergent Change) – ситуация, возникающая из-за нарушения принципа персональной ответственности (SRP), когда для изменения класса существует более одной причины (две или больше осей изменения). То есть, если класс имеет более одной ответственности, каждая из которых может меняться, придется все время править класс, а возможно, и часть его клиентов.

Стрельба дробью (Shotgun Surgery) – внесение какого-то одного изменения приводит к необходимости изменять целый ряд классов.

Группы данных (Data Clumps) - элементы данных, используемые совместно, лучше выделять в классы, или хотя бы для начала в структуры.

Классы данных (Data Class) – это классы, которые содержат только атрибуты (поля), но практически не содержат методов (как структуры), либо содержат только методы доступа к полям. Такие классы полезны на начальных этапах проектирования, но затем на них необходимо возложить дополнительные ответственности.

Завистливые функции (Feature Envy) - методы одного класса активно используют экземпляры других классов, то есть они ближе к другим классам, чем к тому, в котором они определены.

Неуместная близость (Inappropriate Intimacy) – аналогичная ситуация предыдущей, но касается двух и более классов – они слишком активно используют и методы друг друга, и зачастую – те методы и данные друг друга, которые в принципе лучше бы закрыть (либо они вообще имеют доступ к закрытым полям и методам, являясь “друзьями” - friend classes).

Большой класс (Large Class) - раздутый класс, имеющий слишком много методов, то есть сложный интерфейс, зачастую он же имеет слишком много ответственностей.

Ленивый класс (Lazy Class) – класс, который ничего или почти ничего не делает, имеет мало методов, которые редко используются клиентами.

Альтернативные классы с разными интерфейсами (Alternative Classes with Different Interfaces) - имеются два или более класса для выполнения одних и тех же действий, но имеющие разный интерфейс.

Цепочки сообщений (Message Chains) – слишком длинные цепочки делегирования (вызовов методов других классов).

Посредник (Middle Man) – класс, который выполняет только пересылку информации между другими классами.

Отказ от наследства (Refused Bequest) – классы-наследники не используют функциональность родительских классов, иногда и клиенты не

используют функциональность родительских классов, общаясь с наследниками. (Речь не идет о наследовании от интерфейсов).

Теоретическая общность (Speculative Generality) – нарушение принципа “нам это никогда не понадобится” - проявляется в том, что создаются излишне абстрактные классы, параметризуемые классы и другие средства для возможного расширения функциональности в отдаленном будущем [15].

3.3.2 Общая стратегия проведения рефакторинга

Что делать с названными проблемами ? Если коротко то - менять код. А именно – устранять дублирование, выделять и встраивать методы, выделять и встраивать классы, переименовывать переменные, методы, упрощать условные выражения, избавляться от условных операторов и пр.

Чтобы это делать, не боясь ошибиться и без необходимости повторно все отлаживать – нужны модульные тесты.

Согласно К. Беку стратегия рефакторинга предполагает :

- Согласование различий с последующим выделением метода и делегированием (основной подход к устранению дублирования);
- Изоляцию изменений (тут на помощь приходят паттерны и принципы ООП).
- Миграцию данных – для переноса методов и атрибутов из класса в класс.

Наиболее полно шаблоны (приемы) рефакторинга (их часто просто называют рефакторингами) рассмотрены в той же работе М. Фаулера [15].

Самые распространенные рефакторинги : Выделение метода, Замена алгоритма, Переименование переменной, метода, класса, упрощение условного выражения и пр. Выделение метода (Extract method) – один из самых, если не самый распространенный рефакторинг. Разделение

длинного метода на совокупность коротких напоминает разложение на множители (по-английски «factorization»).

Рассмотрим пример - метод вычисления среднего возраста студентов, который обращается к базе данных с помощью средств библиотеки ADO.NET. В таблице 3.1 показан слева код до рефакторинга, справа – после (табличная форма достаточно характерна для представления рефакторинга).

В таблице 3.1 показано выделение методов getReader(), и closeReader(). В результате метод calcAverageAge() уже не зависит от конкретных типов OdbcDataReader/OdbcConnection. В дальнейшем, возможно, потребуется выделить и класс, содержащий данные методы.

Таблица 3.1 Пример рефакторинга с помощью выделения методов

Исходный код	После рефакторинга

public double calcAverageAge() {
double tot_ages = 0;
long cnt = 0;
double res = -1;
OdbcDataReader rdr = null;
OdbcConnection cnn = new
OdbcConnection
("DRIVER=Microsoft Excel Driver
(*.xls);DBQ=stud_list.xls");
OdbcCommand cmd = new
OdbcCommand("select *
from stud_list", cnn);
try {
cnn.Open();
rdr = cmd.ExecuteReader();
while (rdr.Read() {
tot_ages +=
rdr.GetDouble(2);
cnt++;
	OdbcDataReader rdr = null;
	OdbcConnection cnn = null;
	protected DataReader **getReader()**
	{ cnn = new OdbcConnection
	("DRIVER=Microsoft Excel Driver
	(*.xls);DBQ=stud_list.xls");
	OdbcCommand cmd = new
	OdbcCommand("select *
	from stud_list", cnn);
	rdr = null;
	try { cnn.Open();
	rdr = cmd.ExecuteReader();}
	catch { };
	return rdr; }
	protected void **closeReader()** {
	if (rdr) rdr.Close();
	if (cnn) cnn.Close();
	}
	public double calcAverageAge() {

Окончание таблицы 3.1

```
        }
        if (cnt != 0)
           res = Math.Round(
              tot_ages/cnt, 2));
     }
     catch { };
     rdr.Close(); cnn.Close();
     return res;
 }
```

```
double tot_ages = 0;
  long cnt = 0;  double res = -1;
  DataReader  rd = getReader();
  if (!rd) return res;
  while (rd.Read()) {
    tot_ages += rd.GetDouble(2);
    cnt++;
  }
  if (cnt != 0)
     res = Math.Round(
         tot_ages/count, 2));
   closeReader();
   return res;
}
```

3.3.3 Каталог рефакторингов по М. Фаулеру

В книге М. Фаулера [15] приведен каталог, содержащий порядка 70 рефакторингов, разделенных по группам в зависимости от назначения. Заметим, что многие рефакторинги зеркальны друг относительно друга, например, Выделение метода и Встраивание метода. Список рефакторингов по группам приведен в таблице 3.2.

Таблица 3.2 Список рефакторингов из каталога М. Фаулера

Раздел каталога	Рефакторинги
1. Рефакторинги для составления методов	Выделение метода (Extract Method); Встраивание метода (Inline Method); Встраивание временной переменной (Inline Temp); Замена временной переменной вызовом метода (Replace Temp with Query); Введение поясняющей переменной (Introduce Explaining Variable); Расщепление временной переменной (Split Temporary Variable); Удаление присваиваний параметрам (Remove Assignments to Parameters); Замена метода объектом методов (Replace Method with Method Object); Замещение алгоритма (Substitute Algorithm).

Продолжение таблицы 3.2

2. Перемещение функций между объектами	Перемещение метода (Move Method); Перемещение поля (Move Field); Выделение класса (Extract Class); Встраивание класса (Inline Class); Сокрытие делегирования (Hide Delegate); Удаление посредника (Remove Middle Man); Введение внешнего метода (Introduce Foreign Method); Введение локального расширения (Introduce Local Extension).
3. Рефакторинги для организации данных	Самоинкапсуляция поля (Self Encapsulate Field); Замена значения данных объектом (Replace Data Value with Object); Замена значения ссылкой (Change Value to Reference); Замена ссылки значением (Change Reference to Value); Замена массива объектом (Replace Array with Object); Дублирование видимых данных (Duplicate Observed Data); Замена однонаправленной связи двунаправленной (Change Unidirectional Association to Bidirectional); Замена двунаправленной связи однонаправленной (Change Bidirectional Association to Unidirectional); Замена магического числа символической константой (Replace Magic Number with Symbolic Constant); Инкапсуляция поля (Encapsulate Field); Инкапсуляция коллекции (Encapsulate Collection); Замена записи классом данных (Replace Record with Data Class); Замена кода типа классом (Replace Type Code with Class); Замена кода типа подклассами (Replace Type Code with Subclasses); Замена кода типа состоянием/стратегией (Replace Type Code with State/Strategy); Замена подкласса полями (Replace Subclass with Fields).
4. Упрощение условных выражений	Декомпозиция условного оператора (Decompose Conditional); Консолидация условного выражения (Consolidate Conditional Expression); Консолидация дублирующихся условных фрагментов (Consolidate Duplicate Conditional Fragments); Удаление управляющего флага (Remove Control Flag); Замена вложенных условных операторов граничным оператором (Replace Nested Conditional with Guard Clauses); Замена условного оператора полиморфизмом (Replace Conditional with Polymorphism); Введение объекта Null (Introduce Null Object); Введение утверждения (Introduce Assertion).
5. Упрощение вызовов методов	Переименование метода (Rename Method); Добавление параметра (Add Parameter); Удаление параметра (Remove Parameter); Разделение запроса и модификатора (Separate Query from Modifier); Параметризация метода (Parameterize Method); Замена параметра явными методами (Replace Parameter with Explicit Methods); Сохранение всего объекта (Preserve Whole Object); Замена параметра вызовом метода (Replace Parameter with Method); Введение граничного объекта (Introduce Parameter Object); Удаление метода установки значения (Remove Setting Method); Сокрытие метода (Hide Method); Замена конструктора фабричным методом (Replace Constructor with Factory Method);

Окончание таблицы 3.2

	Инкапсуляция нисходящего преобразования типа (Encapsulate Downcast); Замена кода ошибки исключительной ситуацией (Replace Error Code with Exception); Замена исключительной ситуации проверкой (Replace Exception with Test).
6. Решение задач обобщения (рефакторинги, связанные с иерархиями наследования)	Подъем поля (Pull Up Field); Подъем метода (Pull Up Method); Подъем тела конструктора (Pull Up Constructor Body); Спуск метода (Push Down Method); Спуск поля (Push Down Field); Выделение подкласса (Extract Subclass); Выделение родительского класса (Extract Superclass); Выделение интерфейса (Extract Interface); Свертывание иерархии (Collapse Hierarchy); Формирование шаблона метода (Form Template Method); Замена наследования делегированием (Replace Inheritance with Delegation); Замена делегирования наследованием (Replace Delegation with Inheritance).
Крупные рефакторинги (состоят из последовательности более мелких)	Разделение наследования (Tease Apart Inheritance); Преобразование процедурного проекта в объекты (Convert Procedural Design to Objects); Отделение предметной области от представления (Separate Domain from Presentation); Выделение иерархии (Extract Hierarchy).

3.3.4 Краткое описание некоторых рефакторингов

Рассмотрим некоторые рефакторинги из приведенного выше каталога [15]. Для рассмотрения рефакторингов часто используют табличный формат, чтобы сравнивать код до и после преобразования, используем его и мы (таблица 3.3). Примеры и описания для таблицы взяты из книги [15], а обновленные UML-диаграммы - с онлайн каталога рефакторингов М.Фаулера www.refactoring.com [16].

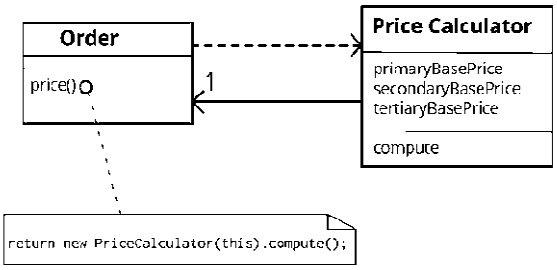
Таблица 3.3 Описание некоторых рефакторингов

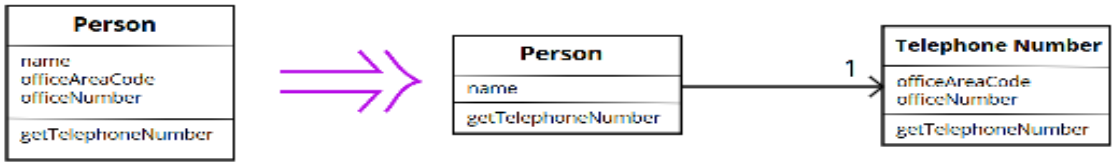
<table>
<tr><td>Имя раздела / шаблона
Проблема / код до рефакторинга</td><td>Решение / код после рефакторинга</td></tr>
<tr><td colspan="2">1 Составление методов</td></tr>
<tr><td colspan="2">1.1 Выделение метода (Extract Method)</td></tr>
<tr><td>Имеется фрагмент кода, который может быть сгруппирован.</td><td>Преобразовать фрагмент в метод, имя которого объясняет его назначение.</td></tr>
<tr><td><pre>
void printOwing() {
 printBanner();
 //print details
 System.out.println("name:"+
     _name);
 System.out.println("amount" +
   getOutstanding());
}
</pre></td><td><pre>
void printOwing() {
   printBanner();
   printDetails(getOutstanding());
}
void printDetails (double outstand)
{
   System.out.println("name:"+
     _name);
   System.out.println("amount" +
     outstand);
}
</pre></td></tr>
<tr><td colspan="2">1.2 Встраивание метода (Inline Method)</td></tr>
<tr><td>Содержание метода очевидно следует из его названия</td><td>Встроить тело метода в вызывающие его модули и удалить метод</td></tr>
<tr><td><pre>
int getRating() {
  return (moreThanFiveLateDeliv ())
    ? 2 : 1;
}
boolean moreThanFiveLateDeliv () {
   return
   _numberOfLateDeliveries > 5;
}
</pre></td><td><pre>
int getRating() {
   return
   (_numberOfLateDeliveries > 5)
   ? 2 : 1;
}
</pre></td></tr>
<tr><td colspan="2">1.3 Введение поясняющей переменной (Explaining Variable)</td></tr>
<tr><td>Имеется сложное выражение.</td><td>Поместить результат выражения или его части во временную переменную с «говорящим» названием.</td></tr>
<tr><td><pre>
if((platform.toUpperCase().indexOf
   ("MAC") > -1) &&
  (browser.toUpperCase().indexOf
   ("IE") > 1)   &&
   wasInitialized() && resize > 0)

{   // do something   }
</pre></td><td><pre>
boolean isMacOs   =
  platform.toUpperCase().indexOf
  ("MAC") > -1;
boolean isIEBrowser =
  browser.toUpperCase().indexOf
  ("IE") > -1;
boolean wasResized = resize > 0;

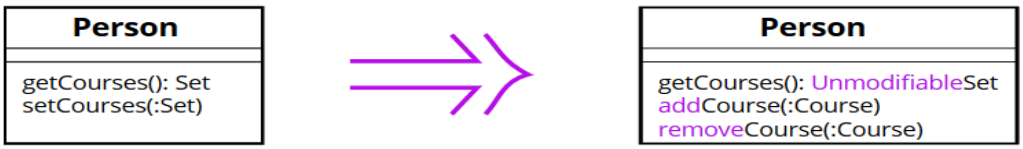
if (isMacOs && isIEBrowser &&
wasInitialized() && wasResized)
{ // do something }
</pre></td></tr>
</table>

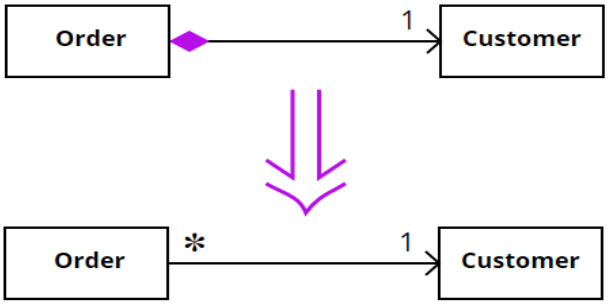
<table>
<tr><td colspan="2">1.4 Замена временной переменной вызовом метода (Replace Temp with Query)</td></tr>
<tr><td>Вы используете временную переменную для хранения результата выражения</td><td>Выделите выражение в метод. Замените все ссылки на данную переменную на вызовы метода. Новый метод может быть использован в новых методах.</td></tr>
<tr><td><pre>
double basePrice =
   _quantity * _itemPrice;
if (basePrice > 1000)
   return basePrice * 0.95;
else
   return basePrice * 0.98;
</pre></td><td><pre>
if (basePrice() > 1000)
   return basePrice() * 0.95;
else
   return basePrice() * 0.98;

double basePrice() {
   return _quantity * _itemPrice;
}
</pre></td></tr>
<tr><td colspan="2">1.5 Замена метода объектом метода (Replace Method with Method Object))</td></tr>
<tr><td>Имеется длинный метод, который использует локальные переменные таким образом, что нельзя применить
Извлечение метода</td><td>Преобразовать метод в объект таким образом, что все локальные переменные становятся полями этого объекта. Затем можно разделить методы на другие методы того же объекта.</td></tr>
<tr><td><pre>
class Order...
    double price() {
      double primaryBasePrice;
      double secondaryBasePrice;
      double tertiaryBasePrice;
            // long computation;
            ...
      }
</pre></td><td> 
</td></tr>
<tr><td colspan="2">1.6 Замещение алгоритма (Substitute Algorithm)</td></tr>
<tr><td>Вы хотите заменить алгоритм другим (например, более понятным)</td><td>Заменить тело метода другим алгоритмом (другой реализацией)</td></tr>
<tr><td></td><td>Например, можно заменить группу значений массивом и использовать циклы, ассоциативные массивы и так далее.</td></tr>
<tr><td colspan="2">2 Перемещение функций между объектами</td></tr>
<tr><td colspan="2">2.1 Перемещение метода (Move Method)</td></tr>
<tr><td>Метод больше связан с другим классом, чем с тем, в котором он определен</td><td>Создать новый метод с тем же содержимым в том классе, с которым он сильнее связан. Затем либо преобразовать тело оригинального метода в простое делегирование обязанностей, либо – удалить исходный метод.</td></tr>
</table>

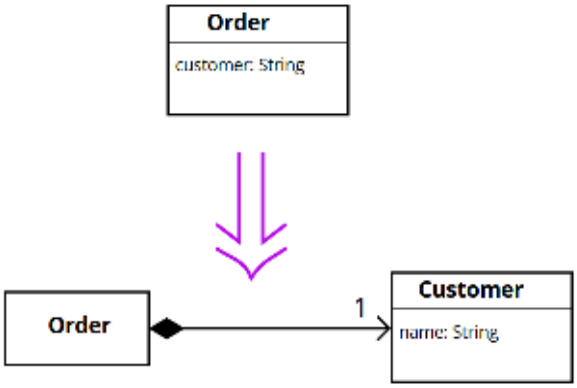
2.2 Перемещение поля (Move Field)	
Поле (атрибут) больше связано с другим классом, чем с тем, в котором оно определено.	Создать новое поле в целевом классе и изменить все связанные с ним методы.
2.3 Выделение класса (Extract Class)	
Имеется один класс, выполняющий работу двух	Создать новый класс и перенести необходимые методы и атрибуты из старого класса в новый
	
2.4 Встраивание класса (Inline Class)	
Класс имеет очень мало обязанностей.	Перенести необходимые методы и атрибуты из класса в другой и удалить класс.
3 Организация данных	
3.1 Инкапсуляция поля (Incapsulate field)	
Класс имеет общедоступное поле.	Сделать поле частным (закрытым) и создать методы доступа.

`public String _name`	`private String _name;`
	`public String getName()`
	`{return _name;}`
	`public void setName(String arg)`
	`{_name = arg;}`

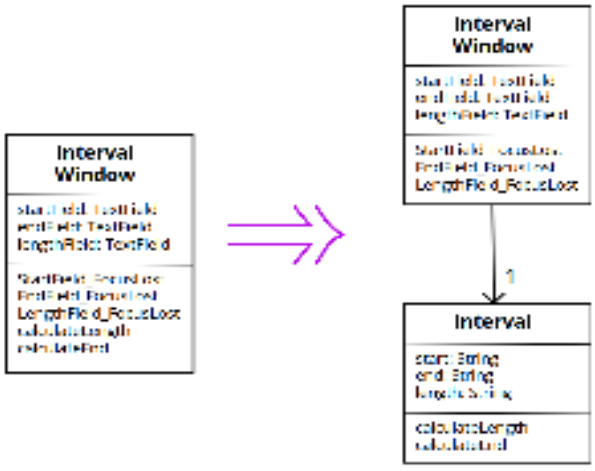
3.2 Инкапсуляция коллекции – Incapsulate Collection	
Метод возвращает коллекцию	Обеспечить возврат коллекции только для чтения и создать методы вставки и удаления.
	
3.3 Замена значения ссылкой (Change Value to Reference)	
Имеется много экземпляров одного и того же класса, которые необходимо заменить на один объект.	Преобразовать объект в ссылку на объект.



3.4 Замена массива / структуры / данных объектом (Replace … with Object).	
Имеется элемент данных, который требует других данных или обладает связанным с ним поведением.	Преобразовать элемент данных в объект.



3.5 Дублирование видимых данных (Duplicate Observed Data).	
Имеются данные предметной области, которые присутствуют только в пользовательском интерфейсе и для них нужен доступ из классов предметной области.	Скопировать данные в объект предметной области. Создать объект-наблюдатель (использовать шаблон Наблюдатель) для синхронизации двух наборов данных.



<table>
<tr><td colspan="2">4 Упрощение условных выражений</td></tr>
<tr><td colspan="2">4.1 Декомпозиция условного оператора (Decompose Conditional).</td></tr>
<tr><td>

```
if (date.before (SUMMER_START)
|| date.after(SUMMER_END))
charge = quantity * _winterRate
+ _winterServiceCharge;
else charge = quantity * _summerRate;
```

</td><td>

```
if (notSummer(date))
charge = winterCharge(quantity);
else charge = summerCharge (quantity);
```

</td></tr>
<tr><td colspan="2">4.2 Консолидация условного выражения (Consolidate Conditional Expression).</td></tr>
<tr><td>Имеется последовательность условий с одинаковым результатом.</td><td>Объединить условия в одно условное выражение и извлечь его в отдельный метод.</td></tr>
<tr><td>

```
double disabilityAmount() {
if (_seniority < 2) return 0;
if (_monthsDisabled > 12) return 0;
if (_isPartTime) return 0;
// compute the disability amount
```

</td><td>

```
double disabilityAmount() {
if (isNotEligableForDisability())
    return 0;
// compute the disability amount
```

</td></tr>
<tr><td colspan="2">4.3 Консолидация дублирующихся условных фрагментов (Consolidate Duplicate Conditional Fragments).</td></tr>
<tr><td>Один и тот же фрагмент кода присутствует во всех ветвях условного оператора.</td><td>Перенести повторяющийся фрагмент вовне условного оператора.</td></tr>
<tr><td>

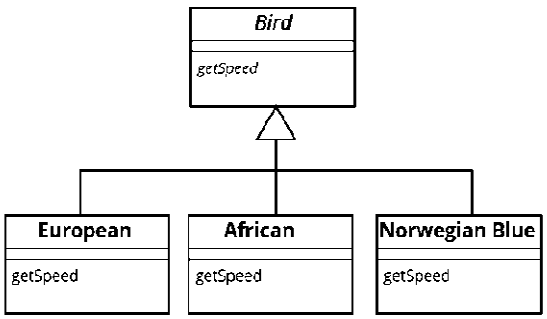
```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

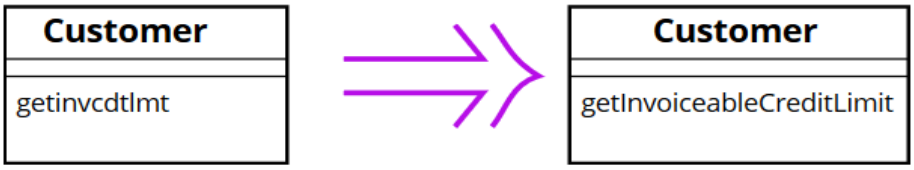
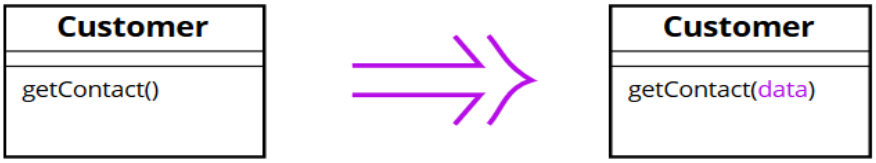
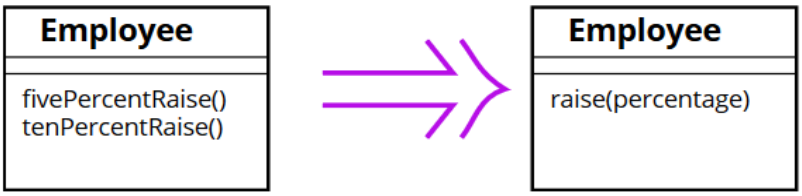
</td><td>

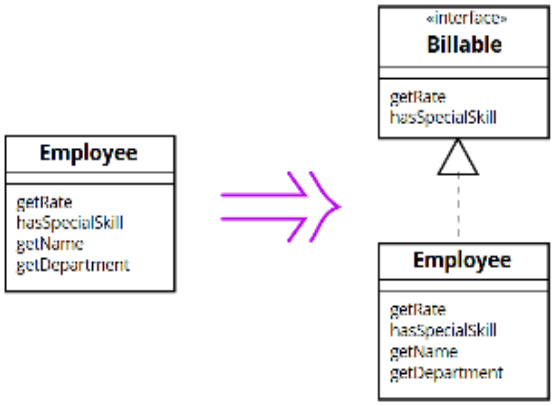
```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;
send();
```

</td></tr>
<tr><td colspan="2">4.4 Замена условного оператора на полиморфизм (Replace Conditional with Polymorphism).</td></tr>
<tr><td>Имеется оператор выбора (или ветвление) который выбирает различное поведение, зависящее от типа объекта.</td><td>Перенести различные ветви оператора в переопределяемый метод подкласса. Сделать исходный метод абстрактным.</td></tr>
<tr><td>

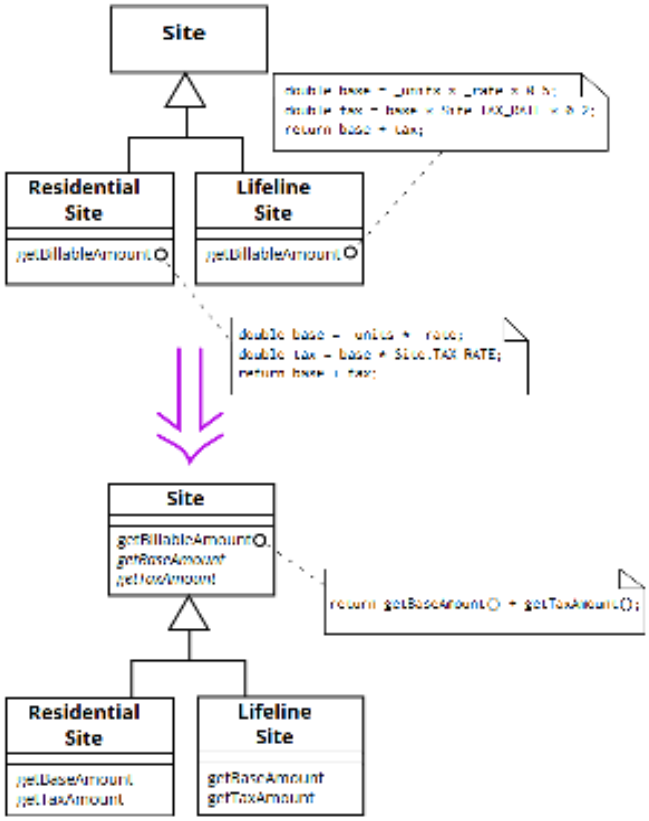
```
double getSpeed() {
   switch (_type) {
      case EUROPEAN:
         return getBaseSpeed();
      case AFRICAN:
         return getBaseSpeed() -
         getLoadFactor() *
         _numberOfCoconuts;
      case NORWEIGIAN_BLUE:
         return (_isNailed) ?
         0 : getBaseSpeed(_voltage);
   }
throw new RuntimeException ("Should be
unreachable");
}
```

</td><td> 
</td></tr>
</table>

<table>
<tr><td colspan="2">5 Упрощение условных выражений</td></tr>
<tr><td colspan="2">5.1 Переименование метода (Rename Method).</td></tr>
<tr><td>Имя метода не раскрывает его назначение.</td><td>Изменить имя метода.</td></tr>
<tr><td colspan="2"> 
</td></tr>
<tr><td colspan="2">5.2 Добавление и удаление параметра (Add / remove Parameter).</td></tr>
<tr><td>Метод нуждается в дополнительной информации от вызывающего объекта.</td><td>Добавить параметр в метод.</td></tr>
<tr><td colspan="2"> 
</td></tr>
<tr><td colspan="2">5.3 Параметризация метода (Parameterize method).</td></tr>
<tr><td>Метод нуждается в дополнительной информации от вызывающего объекта.</td><td>Добавить параметр в метод.</td></tr>
<tr><td colspan="2"> 
</td></tr>
<tr><td colspan="2">5.4 Замена конструктора фабричным методом (Replace Constructor With Factory Method)</td></tr>
<tr><td>Вы хотите делать что-то более сложное, чем простое создание при создании объекта. (А вернее сказать – есть необходимость спрятать детали создания объекта).</td><td>Заменить конструктор фабричным методом (паттерн проектирования Фабрика).</td></tr>
<tr><td><pre>Employee (int type) {
   _type = type;
}</pre></td><td><pre>static Employee create(int type) {
   return new Employee(type);
}</pre></td></tr>
<tr><td colspan="2">5.5 Сокрытие метода (Hide Methode)</td></tr>
<tr><td>Метод класса не используется другими классами.</td><td>Сделать метод закрытым.</td></tr>
</table>

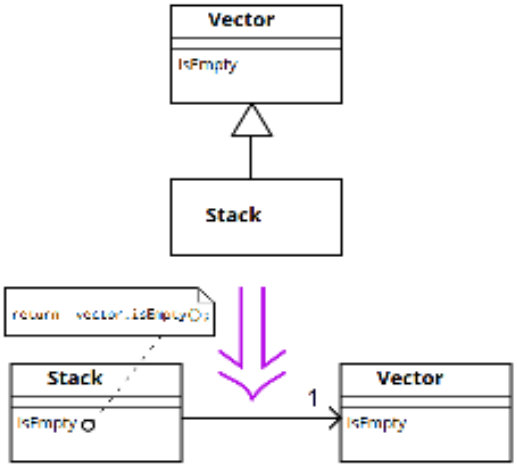
6 Решение задач обобщения	
6.1 Подъем поля / метода (Pull-Up Filed / Method)	
Два подкласса имеют общее поле.	Переместить поле в родительский класс
6.2 Спуск поля / метода (Pull Down Filed / Method)	
Общее поле необходимо лишь в одном классе.	Переместить поле в производный класс
6.3 Выделение подкласса (Extract Subclass)	
Класс имеет свойства, которые используются только у каких-то конкретных экземпляров или у типа экземпляров.	Создать подкласс для этой группы экземпляров.
6.4 Выделение родительского класса (Extract Superclass)	
Имеются два класса с похожими свойствами.	Создать родительский класс, выделив туда общие свойства этих классов.
6.5 Выделение интерфейса (Extract Interface)	
Несколько клиентов используют одно и то же подмножество интерфейса класса, или два класса имеют пересекающуюся часть интерфейса.	Выделить это подмножество интерфейса в отдельный интерфейс.
	
6.6 Свертывание иерархии (Collapse Hierarchy)	
Родительский класс и класс наследника не слишком отличаются.	Объединить классы в один.
6.7 Формирование шаблона метода (Form Template Method)	
Имеются два метода в подклассах, которые выполняют похожие шаги в похожем порядке, хотя детали различны.	Постепенно сделать эти методы одинаковыми, преобразуя шаги в методы с одинаковой сигнатурой. Затем можно выполнить Подъем метода.

Окончание таблицы 3.3



6.8 Замена наследования делегированием и наборот (Replace Inheritance with Delegation vice versa).

Наследник класса использует только часть интерфейса родителя или не хочет наследовать его данные.	Создать поле для родительского класса, преобразовать методы для выполнения делегирования этому классу и устранить наследование.



3.4. Разработка через тестирование (TDD) как концепция

Концепция TDD выражается принципом «Вначале тесты». Мы начинаем написание кода модуля с написания модульного теста. Разработка модуля состоит из коротких циклов, описываемых метафорой «светофора»: 1) создание теста, код сначала возможно даже не компилируется из-за отсутствия нужного кода в самом модуле («желтый свет»); 2) код компилируется, но тест не проходит («красный свет»); 3) кодирование основного модуля до выполнения теста («зеленый свет»); 4) выполняем рефакторинг, если нужно, 5) пишем новый тест, он снова не проходит (цикл «поломать, починить, улучшить», «желтый, красный зеленый, рефакторинг») и так далее.

Прежде чем написать модуль, вы пишете тестовый случай (TestCase), который соответствует интерфейсу модуля, который вы разрабатываете. Если вы еще не знаете интерфейс, но имеете CRC карточку создаваемого класса (на CRC карточке описан класс, его ответственность и взаимодействие с другими классами – Class-Responcibility-Cooperation), вы с помощью тестов формируете интерфейс, затем – реализацию интерфейса, затем – улучшаете код до тех пор, пока он не начнет вас устраивать. Зачастую именно необходимость тестирования даже существующего кода наталкивает на мысль о более удачном дизайне. Например, это позволяет разделить (выделить) интерфейсы, выполнить декомпозицию и т.д.

3.5 Небольшой пример разработки через тесты

Тестируем метод для поиска максимального элемента в массиве целых чисел на языке C# (пример носит чисто учебный характер, так как в библиотечном типе массива уже есть такой метод) [14]:

```
static int Largest(int[] list);
```

Как составить список тестов ? Первое, что приходит в голову:

[7, 8, 9] -> 9.

Авторы предлагают также проверить влияние очередности :

[7, 8, 9] -> 9,

[8, 9, 7] -> 9,

[9, 7, 8] -> 9.

Так же разумно проверить работу метода при дублировании максимальных чисел:

[7, 9, 8, 9] -> 9,

поиск в массиве из одного элемента :

[1] -> 1,

а также – поиск среди отрицательных чисел :

[-9, -8, -7] -> -7.

Создаем код класса по шаблону Очевидная реализация (такой резкий старт не совсем соотносится с TDD) :

```
public class Cmp {
        public static int Largest(int[] list) {
          int index, max=Int32.MaxValue;
          for (index = 0; index < list.Length-1; index++) {
                if (list[index] > max) {     max = list[index];  }
          }
          return max;
        }
}
```

Теперь создаем первый тест (используем NUnit) :

```
using NUnit.Framework;
[TestFixture]
public class TestLargest {
        [Test]
        public void LargestOf3() {
```

```
            Assert.AreEqual(9, Cmp.Largest(new int[] {8,9,7}));
        }
    }
```

Запускаем с помощью NUnit TestRunner, получаем красную полосу и сообщение:

```
Failures:
1) TestLargest.LargestOf3 :
expected:<9>
but was:<2147483647>
at TestLargest.LargestOf3() in c:ntestlargest.cs:line 6
```

Исправляем ошибку – вместо max=Int32.MaxValue нужно max = 0.

Добавляем в тест еще две строчки :

```
Assert.AreEqual(9, Cmp.Largest(new int[] {8,9,7}));
Assert.AreEqual(9, Cmp.Largest(new int[] {9,8,7}));
Assert.AreEqual(9, Cmp.Largest(new int[] {8,7,9}));
```

Запускаем, получаем опять ошибку:

```
Failures:
1) TestLargest.LargestOf3 :
expected:<9>
but was:<8>
at TestLargest.LargestOf3() in c:ntestlargest.cs:line 5
```

Исправляем: index < list.Length. Тест проходит и нам приходит мысль о рефакторинге – использовать итератор foreach, чтобы сделать код нагляднее :

```
    foreach (int num in list) {
        if (num > max) {    max = num;  }
    }
```

После этого небольшого рефакторинга мы запускаем все тесты, и они проходят (отлаживать код заново не нужно).

Проверка с одним значением также проходит, и мы добавляем тест для проверки отрицательных значений:

```
[Test]
public void TestNegative() {
        int [] negList = new int[] f-9, -8, -7g;
        Assert.AreEqual(-7, Cmp.Largest(negList));
}
```

Тест не проходит (мы опять не так инициализировали наше максимальное значение): вместо max=Int32.MaxValue нужно max=Int32.MinValue. Исправляем. Тест проходит. Тут мы вспоминаем о случае пустого массива. А что возвращать в данном случае из функции, ведь у пустого массива нет максимума ? Решаем генерировать исключение (тест помогает нам принять проектное решение):

```
[Test, ExpectedException(typeof(ArgumentException))]
public void TestEmpty() {
        Cmp.Largest(new int[] {});
}
```

Теперь создаем код для выполнения этого теста :

```
if (list.Length == 0) { throw new ArgumentException("Empty list"); }
```

Все тесты проходят. Все ли тесты мы создали? Наверное, нет. При желании мы можем провести исчерпывающее тестирование.

4 ПОНЯТИЕ О ПАТТЕРНАХ ПРОЕКТИРОВАНИЯ

Паттерн (шаблон) проектирования – именованное описание проблемы и ее решения (решений), которые можно применить при разработке других систем. Это именованная пара "проблема / решение", содержащая рекомендации для применения в различных контекстах. Термин «паттерн» является дословной транскрипцией английского «pattern», и, хотя по-русски более привычный перевод для «pattern» это «шаблон», среди специалистов и в специальной литературе более популярен именно термин «паттерн». Возможно, это призвано указать на варианты перевода «pattern» как «образец», «модель», «структура», «стреотип», что ближе к pattern, в отличие, например, от «template» («шаблон», «трафарет»). Далее мы будем использовать оба термина («паттерн» и «шаблон»), подразумевая под ними именно образцы решений («patterns»).

При создании программного обеспечения используются шаблоны разного уровня :

1. Архитектурные шаблоны (например, Layers, Service Layer и другие)
2. Шаблоны проектирования классов (например, GoF, GRASP [3], другие).
3. Идиомы – низкоуровневые шаблоны, ориентированные на конкретный язык или программную платформу (например, реализация шаблона Наблюдатель в .NET/C# или в Qt).

Если рассматривать шаблон как набор рекомендаций для определенных ситуаций, то можно выделить помимо шаблонов проектирования:

- шаблоны анализа;
- шаблоны планирования и пр.

– шаблоны тестирования;

– шаблоны рефакторинга и другие.

Можно сказать, что шаблоны тестирования и рефакторинга мы рассмотрели ранее в данном пособии, хотя они и менее формализованы, чем паттерны проектирования.

Также шаблоны могут разделяться по типам приложений, для которых они предназначены :

– шаблоны построения корпоративных приложений;

– шаблоны для веб-приложений;

– шаблоны для мобильных приложений;

– шаблоны для систем реального времени;

– шаблоны для многопоточных и распределенных приложений,

а также по подсистемам приложений :

– шаблоны построения интерфейса пользователя;

– шаблоны построения слоя предметной области;

– шаблоны проектирования и взаимодействия с базой данных и так далее.

При этом зачастую одни и те же шаблоны и рекомендации могут относиться к разным классам в зависимости от текущей точки зрения, например, шаблон Контроллер (один из шаблонов GRASP) можно рассматривать как шаблон проектирования классов, как архитектурный шаблон, как часть концепции Модель-Вид-Контроллер (MVC). Тот же шаблон MVC можно рассматривать более узко как шаблон организации взаимодействия с интерфейсом пользователя, как архитектурный шаблон для веб-приложений, либо в целом как общий архитектурный шаблон.

4.1 Понятие об архитектурных шаблонах. Шаблон Слои. Трехуровневая и многоуровневая организация приложения

Один из самых известных архитектурных шаблон – это шаблон Слои (Layers, К. Ларман, [3]). Примеры архитектурных шаблонов, а также шаблонов других классов, которые можно, на наш взгляд, в некоторой степени отнести к архитектурным :

1) Клиент-сервер;
2) Слои (Layers);
3) Слой служб (Service Layer);
4) Фасад (Facade);
5) Контроллер (Controller, шаблон GRASP и часть MVC);
6) Принцип Model-View Separation и принцип MVC;
7) Информационный эксперт (шаблон GRASP);
8) Сильное зацепление и Слабое связывание (шаблоны GRASP);

и другие.

Первые три шаблона являются собственно архитектурными, остальные – до некоторой степени влияют на общую архитектуру.

В целом все архитектурные шаблоны предлагают какие-то решения, относящиеся к архитектурному представлению системы как совокупности крупных подсистем, формируют взгляд на систему с точки зрения высокоуровневого проектирования, с «высоты птичьего полета».

Наиболее известным шаблоном такого типа является Клиент-сервер, который предполагает разделение системы на две части, клиента и сервера, последний часто может обслуживать множество клиентов. Часто этот шаблон возникает в контексте описания построения систем, использующих базы данных (БД). Сам шаблон является достаточно общим и не специфицирует особенности реализации этих двух подсистем, за исключением некоторых уточнений относительно клиента в виде «тонкий клиент», «толстый клиент», «терминальный клиент» и т.п.

Несколько более детализированным является архитектурный шаблон Слои (Layers), описанный, в частности, в книге К. Лармана [3].

Принципы шаблона Слои [3]:

1. «Организовать крупномасштабные структурные элементы системы в отдельные уровни со взаимосвязанными обязанностями таким образом, чтобы на нижнем уровне располагались низкоуровневые службы и службы общего назначения, а на более высоких уровнях – объекты уровня логики приложения.

2. Взаимодействие и связывание уровней происходит сверху вниз. Нужно избегать связывания объектов снизу вверх» [3].

Концепции уровней или слоев приложения легче всего проиллюстрировать на примере информационных систем, использующих базы данных (БД).

Традиционная двухуровневая система «клиент-сервер» подразумевает наличие интерфейса пользователя и слоя доступа к данным, или точнее – к серверу БД. Собственно логика, имеющая отношение к предметной области приложения, располагается в одном из этих слоев. При этом объем этой логики сравнительно невелик, поскольку основная логика сосредоточена на сервере БД. В некоторых случаях даже подразумевается, что один уровень – это все клиентское приложение, в котором не выделяются отдельно слои интерфейса пользователя (UI), логики приложения и код для доступа к данным, а второй слой – это собственно сам сервер БД, на котором размещаются хранимые процедуры и виды, реализующие основную логику приложения с помощью SQL или его скриптовых расширений (T-SQL, PL/SQL и др.). В целом эти оба варианта страдают отсутствием четкой локализации функциональности приложения в коде программы и, соответственно, недостаточной модульностью, гибкостью, переносимостью, затруднением повторного использования кода, затруднением тестирования приложения и т.д.

Присутствует зависимость решения от платформы и средства и языка разработки. Другая проблема состоит в реализации логики, подчас сложной, на декларативном языке SQL или на скриптовых языках, не поддерживающих объектные возможности. По мере развития информационной системы и усложнения бизнес-правил и модели предметной области затрудняется развитие и сопровождение системы, даже может снижаться производительность. Наконец, если рассматривать второй вариант, в котором все клиентское приложение однослойно, то как правило, такое приложение вообще не является модульным, объектным, содержит массу повторяющихся фрагментов, тяжело модифицируется и так далее.

Альтернативой такому варианту являются многослойные, в частности, трехслойные системы.

Принцип трехуровневой архитектуры для информационных систем достаточно прост: выделяется уровень представления, уровень прило-жения и уровень данных (технический слой, или уровень работы с базой данных). При этом уровни четко разделены, зависимость между уровнями направлена сверху вниз, в результате можно заменять источники данных, не трогая интерфейс, или заменять интерфейс, не трогая средний уровень логики. Повышается потенциал повторного использования кода (подсистем приложения), облегчается тестирование, облегчается возможность создания распределенных приложений и так далее.

Пример архитектуры трехуровневого приложения из области финансов с двумя версиями интерфейса пользователя на языке и с использованием библиотек Java приведен на рис. 4.1 [3].

Шаблон Layers в общем случае предполагает наличие не трех, а большего числа слоев, степень специфичности которых снижается сверху вниз, а область применения, наоборот – повышается. Хорошо согласуется с этим шаблоном и развивает его принцип инверсии зависимостей (DIP),

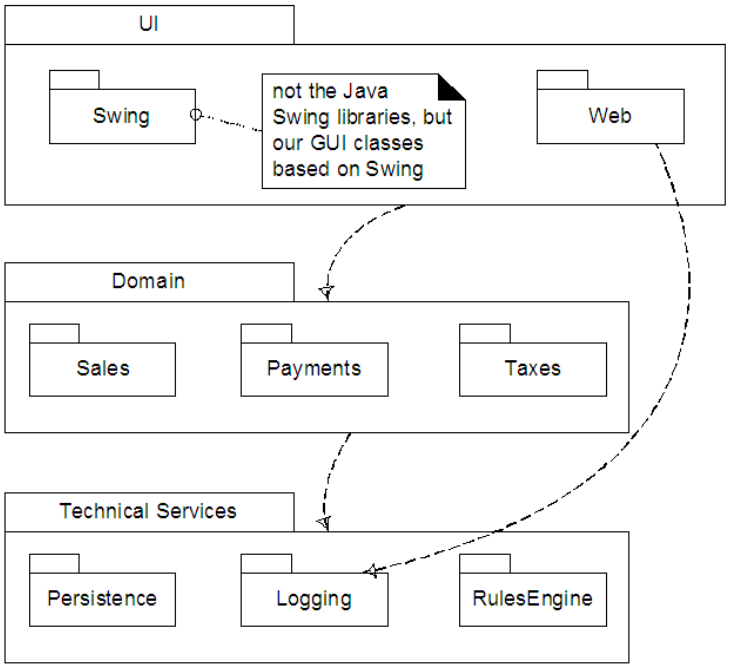


Рис. 4.1 – Пример трехслойной модели приложения в паттерне Слои

который говорит о том, что в ООП стараются преобразовывать зависимость слоев сверху вниз в зависимость вышестоящих слоев от интерфейсов слоев более низкого уровня, которые реализуются нижестоящими слоями, точнее, отдельными прослойками-адаптерами. В результате нижестоящие слои начинают зависеть от вышестоящих в плане предоставляемой им функциональности (через интерфейс). Основной принцип шаблона Слои при этом не нарушается, так как от вышестоящего слоя зависят только прослойки, реализующие требуемые интерфейсы, а не основные подсистемы нижестоящих слоев. С другой стороны, скажем, слой логики приложения может сам требовать от слоя представления реализовать требуемые ему интерфейсы, сохраняя общее направления зависимости слоев сверху вниз неизменным, как, например, реализуется в шаблоне MVP.

4.2 Паттерны проектирования GoF

Выше мы выделили архитектурные паттерны, паттерны анализа, паттерны тестирования, паттерны рефакторинга и другие Паттерны проектирования классов относят к уровню микропроектирования, хотя некоторые из них вполне приемлемы и на уровне архитектурного проектирования, как было отмечено в начале данного раздела.

Паттерны проектирования (design patterns) - специальные схемы для уточнения структуры подсистем или компонентов программной системы и отношений между ними.

Паттерны проектирования описывают общую структуру взаимодействия элементов программной системы, которые реализуют исходную проблему проектирования в конкретном контексте. Наиболее известными паттернами этой категории являются паттерны GoF (Gang of Four), названные в честь Э. Гаммы, Р. Хелма, Р. Джонсона и Дж. Влиссидеса, которые систематизировали их и представили общее описание в своей книге [17]. Паттерны GoF включают в себя 23 паттерна. Эти паттерны не зависят от языка реализации, но их реализация зависит от области приложения.

4.2.1 Назначение паттернов проектирования

Как и рефакторинг, паттерны прежде всего направлены на защиту проекта от изменений путем сокрытия (инкапсуляции) деталей реализаций одних подсистем приложения от других. Для удобства изложения будем пользоваться распространенной терминологией, включающей клиентский код и код сервера. Сразу оговорим, что речь не идет в данном случае о разных программах (клиенте и сервере), о шаблоне «клиент-сервер» и даже далеко не всегда о клиентской и серверной подсистемах одного

приложения. В данном случае клиент – это один класс (или группа классов), который использует для своих целей другой класс (или группу классов) – сервер. Тогда можно сказать, что одной из важных целей паттернов проектирования является инкапсуляция деталей реализации серверов от их клиентов.

Для этого используется делегирование, введение абстракций и наследование от них, инкапсуляция и полиморфизм. Все это в конечном итоге можно отнести к различным способам введения дополнительной косвенности - вместо того чтобы передавать клиенту что-то, мы указываем, как это что-то можно получить. Для этого используется либо интерфейс, либо – класс-посредник, либо совокупность классов. Просто в каждом конкретном случае дается более точное описание ситуации, описание того, что мы скрываем, описание того, как организовать это сокрытие. Соответственно и разделы каталога шаблонов GoF соответствуют различным целям сокрытия – паттерны структурирования скрывают структуры и интерфейсы классов, их взаимосвязь, паттерны поведения – поведение классов, паттерны создания – детали создания.

Рефакторинг также предлагает дополнительную косвенность, но прежде всего – за счет выделения методов и делегирования. Затем он также предлагает решение задач обобщения, использование интерфейсов и т.п. Просто рефакторинг, с одной стороны, более общая практика, с другой стороны – более эволюционная, паттерны больше похожи на список рецептов для более специализированных случаев.

Паттерны позволяют ввести общеупотребимую терминологию для проектных решений, понятную всем разработчикам, которые с ними знакомы.

Таким образом, цели паттернов проектирования :

- защита (инкапсуляция) изменений;
- разделение обязанностей (разделение работы);

- возможности масштабирования;

- использование единой проектной терминологии.

Несмотря на общепризнанную полезность паттернов проектирования нельзя не отметить и их критику. В частности, существует точка зрения, что паттерны «разрушают принцип ООП» в том смысле, что часто они вносят в объектную модель технические классы, которым нет аналогов в предметной области задачи, чем нарушают наглядность модели программных классов, что отчасти верно. С другой стороны, нужно понимать, что за каждым паттерном стоит большой опыт объектного проектирования многих специалистов, который позволяет существенно экономить ресурсы времени разработчиков, позволяя им быстрее получать более эффективные и грамотные решения.

Что касается наглядности объектной модели и соответствии ее реальной жизни, в качестве полезной практики при использовании паттернов, как и вообще при объектном проектировании можно рекомендовать персонификацию объектов (классов). Под персонификацией понимают представление сущностей, природных явлений и сил в образе действующих лиц или признание за ними человеческих свойств, что помогает лучше понять и представить себе взаимодействие объектов в программе, в том числе и тех, которые не соответствуют реальным сущностям предметной области.

Другое критическое замечание по поводу паттернов проектирование касается не их самих, а их изучения. Существует точка зрения, что паттерны нельзя изучить по учебнику, а можно только систематизировать с помощью них свой собственный опыт, уже набив свои собственные шишки и по-настоящему поняв те проблемы, которые призваны решить паттерны. Это тоже отчасти верно, но, с другой стороны, так можно сказать о многих аспектах деятельности человека и приобретаемых им знаниях, в том числе в программировании. Этот дискуссионный вопрос

относится к числу общих традиционных вопросов о соотношении практики и теории, собственного и чужого опыта, своих и чужих ошибках, содержании образования в целом и так далее.

4.2.2 Общая характеристика каталога паттернов GoF

Каталог паттернов проектирования GoF можно разделить на три основных раздела по назначению (по тому, детали чего инкапсулирует или скрывает паттерн) :

- паттерны создания;
- паттерны структурирования;
- паттерны поведения.

Кроме того, паттерны разделяются по уровню: уровня объектов (основаны на делегировании промежуточным объектам, в роли которых часто выступают интерфейсы) и уровня классов (основаны на наследовании, чаще всего от абстрактных классов).

В таблице 4.1 приведены все 23 паттерна GoF, разделенные на 6 подгрупп по назначению и уровням. Один из паттернов – адаптер – имеет две модификации: адаптер объектов и адаптер классов. Наиболее часто используемых паттернов GoF - примерно 13 : Абстрактная Фабрика, Адаптер, Команда, Композит, Итератор, Фасад, Наблюдатель, Заместитель, Одиночка, Состояние, Стратегия, Шаблонный метод, Фабричный метод. Они выделены в таблице жирным шрифтом.

Сами авторы рекомендуют начинать изучение шаблонов со следующих: Адаптер, Декоратор, Композит, Наблюдатель, Стратегия, Фабрика, Фабричный метод, Шаблонный метод. Сюда можно добавить еще Итератор, Команду, Фасад.

Таблица 4.1 Структура каталога паттернов GoF

		Назначение паттернов		
		Создания	Структурирования	Поведения
Уро-вень	Классы	**Фабричный метод (Factory Method)**	**Адаптер (Adapter)**	Интерпретатор (Interpreter) **Шаблонный метод (Template Method)**
	Объекты	**Абстрактная фабрика (Abstract Factory)** Строитель (Builder) Прототип (Prototype) **Одиночка (Singleton)**	**Адаптер (Adapter)** Мост (Bridge) **Компоновщик (Composite)** Декоратор (Decorator) **Фасад (Facade)** **Заместитель (Proxy)**	Цепочка ответственности (Chain of Responsibility) **Команда (Command)** **Итератор (Iterator)** Посредник (Mediator) Хранитель (Memento) Приспособленец (Flyweight) **Наблюдатель (Observer)** **Состояние (State)** **Стратегия (Strategy)** Посетитель (Visitor)

В следующих пунктах рассмотрим три основных раздела каталога (по назначению паттернов) подробнее.

4.2.3 Паттерны структурирования

Паттерны структурирования призваны скрыть структуру одного объекта (Адаптер), группы объектов (Фасад), структуру сложных объектов-агрегатов (Компоновщик), дополнительные функции, расширяющие целевой объект (Заместитель, Декоратор), либо иерархию (Мост) от клиентов. Список паттернов структурирования приведен в таблице 4.2. Далее рассмотрим часть паттернов структурирования подробнее.

Таблица 4.2. Список структурных паттернов проектирования GoF

№	Название паттерна	Перевод	Назначение паттерна
1	Adapter (Wrapper)	Адаптер (Обертка)	Преобразует существующий интерфейс класса в другой интерфейс, который понятен клиентам. При этом обеспечивает совместную работу классов, невозможную без данного паттерна из-за несовместимости интерфейсов.
2	Bridge	Мост	Отделяет абстракцию класса от его реализации, благодаря чему появляется возможность независимо изменять то и другое.
3	Composite	Компоновщик	Группирует объекты в иерархические структуры для представления отношений типа "часть-целое", что позволяет клиентам работать с единичными объектами так же, как с группами объектов.
4	Decorator	Декоратор	Применяется для расширения имеющейся функциональности и является альтернативой порождению подклассов на основе динамического назначения объектам новых операций.
5	Facade	Фасад	Предоставляет единый интерфейс к множеству операций или интерфейсов в системе на основе унифицированного интерфейса для облегчения работы с системой.
6	Proxy	Заместитель	Подменяет выбранный объект другим объектом для управления контроля доступа к исходному объекту.

4.2.3.1 Адаптер

Проблема, которую решает паттерн – преобразовать интерфейс класса в другой (целевой) интерфейс, который ожидает клиент. Позволяет совместно работать классам, которые не могли бы этого делать без адаптера из-за несовместимых интерфейсов. (т.е. нужно а) согласовать интерфейсы; б) спрятать интерфейс сложного класса). Также известен как Обертка (Wrapper).

Мотивация использования. Часто библиотечный класс, разработанный для повторного использования, не может использоваться в конкретном приложении из-за неудобного интерфейса.

Применимость. Адаптер можно использовать, когда:

- необходимо использовать существующий класс, но его интерфейс не совпадает с тем, который нужен (удобен);

- Вы хотите создать класс для повторного использования, который взаимодействует с классами с неизвестным интерфейсом.

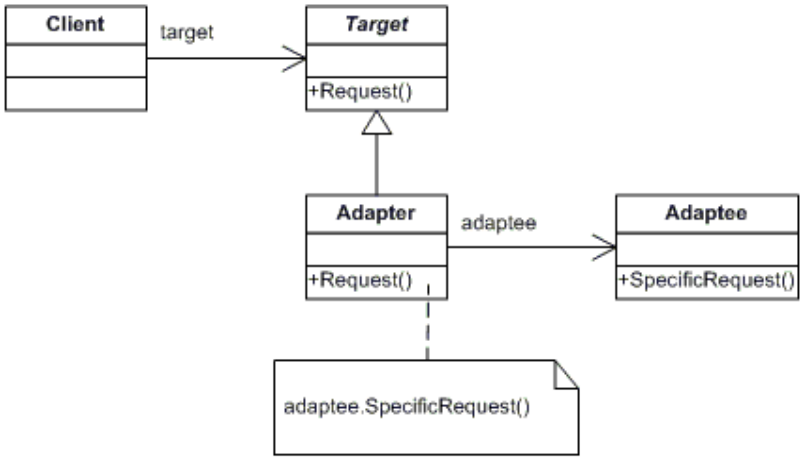


Рис. 4.2. Структура паттерна Адаптер объектов

Структура паттерна представлена на рис. 4.2 (здесь и далее структуры паттернов представлены в виде диаграмм классов UML, приведенных на сайте [18]). Участники паттерна: Client – класс-клиент, нуждающийся в интерфейсе Target. Adaptee – имеющийся класс, содержит требуемую клиенту функциональность, но обладает другим интерфейсом. Adapter – центральный участник паттерна – класс-адаптер, который мы вводим для согласования двух интерфейсов (Adaptee и Target).

Как и во всех паттернах, в данном случае и названия участников паттерна, и тем более – названия их методов не являются обязательными, хотя обычно разработчики стараются, по крайней мере для ключевых участников (в данном случае – для Adapter) использовать в названии

конкретного адаптера термин Adapter или Wrapper, например, DataWrapper или CalculatorAdapter.

Похожие паттерны и связь с другими паттернами. Мост похож на Адаптер, но имеет другое назначение (для проектирования новых классов, адаптер – для адаптирования существующего). Декоратор – расширяет существующий класс, не меняя его интерфейс. Декоратор поддерживает рекурсивную композицию, а адаптер – нет. Заместитель замещает существующий объект для контроля доступа к нему и не меняет его интерфейс.

Особенности адаптера. Одной из особенностей является наличие двух версий паттерна : адаптер объектов и адаптер классов (в последнем случае вместо делегирования классу Adaptee используется наследование от него). Адаптер классов встречается реже, как и в целом наследование в ООП стараются применять реже, чем делегирование. Одним из любопытных частных случаев применения паттерна Адаптер классов является модульное тестирование закрытых методов класса. В этом случае закрытые (private) методы делаются защищенными (protected), создается наследник, который делегирует этим методам в своих открытых методах для целей тестирования модульным тестом. Этот наследник играет роль Адаптера классов.

Другая особенность паттерна Адаптер состоит в том, что он часто применяется без явного выделения интерфейса Target, который неявно, конечно, существует, так как именно его реализует адаптер для клиента. Более того, адаптер может и сам не существовать в виде отдельного класса, первоначально он может реализовываться просто как отдельный метод Клиента.

4.2.3.2 Фасад

Проблема, которую решает паттерн – скрыть от клиента несколько взаимодействующих классов за общим интерфейсом одного класса-фасада. Также известен как Шлюз (хотя под Шлюзом часто понимают фасад исключительно для внешних подсистем).

Мотивация использования Фасада достаточно очевидна – мы хотим скрыть от клиентов детали реализации какой-то подсистемы, какие классы в нее входят и как они взаимодействуют. Без Фасада пришлось бы наделять всех клиентов знанием обо всех классах подсистемы, что, конечно, сильно усложняет всю картину.

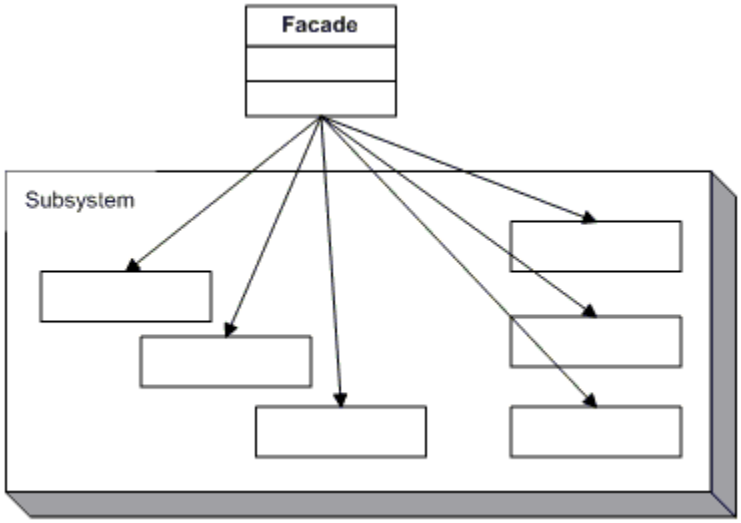


Рис. 4.3. Структура паттерна Фасад

Структура паттерна представлена на рис. 4.3. Собственно тут два участника : Фасад (Facade) и скрываемая за ним Подсистема (subsystem).

Фасад похож на паттерн Адаптер, но в отличие от него скрывает от клиента интерфейс не одного, а сразу нескольких классов, как и их взаимосвязи. Также в отличие от Адаптера, как и Мост он часто вводится на этапе проектирования, а не постфактум.

Одной из особенностей фасада является отсутствие класса-интерфейса в структуре паттерна, хотя его легко можно добавить, выделив

интерфейс Фасада и реализовав его для конкретной подсистемы, что часто и делают на практике.

4.2.3.3 Компоновщик

Проблема, которую решает данный паттерн – представить структуры типа «часть-целое» таким образом, чтобы их можно было обрабатывать единообразно.

Мотивация использования паттерна – удобное представление иерархических вложенных структур с общими свойствами и методами, содержащих как простые, так и составные объекты. Например, в книге GoF приводится довольно наглядный пример текстового процессора. Можно предложить не менее наглядный пример обработки фигур в графическом редакторе, когда требуется единообразно обрабатывать (например, перемещать, менять заполнение, цвет, начертание и др.) как простые графические примитивы, так и составные графические объекты, в том числе вложенные друг в друга.

Структура паттерна представлена на рис. 4.4. Клиенту Client предоставляется интерфейс/абстрактный класс Component, который не только включает общую для всех членов иерархии операцию Operation(), но и методы работы с дочерними (вложенными) объектами Add(), Remove(), GetChild() (напоминаем, что эти и другие конкретные названия вовсе не обязательны при реализации паттерна). От абстрактного Component наследуют листовой Leaf (он не нуждается в методах работы с вложенными объектами, эти вырожденные методы могут реализовываться по умолчанию в Component) и собственно составной Composit, содержащий список объектов children с интерфейсом Component. Как показано на рисунке 4.4, основной метод Operation() в Composit вызывает одноименный метод у всех вложенных в него объектов-компонентов.

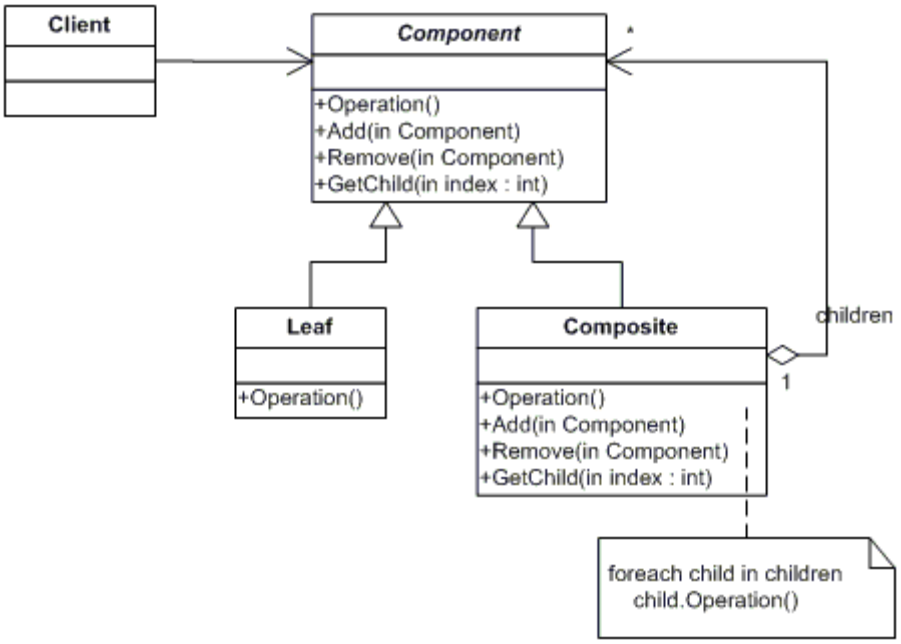


Рис. 4.4. Структура паттерна Компоновщик

Паттерн Компоновщик (или Композит) часто используется вместе с паттернами Итератор и Фабричный метод (описаны далее) для организации инкапсуляции коллекции children, ее перебора и удобной работы с ней в целом.

4.2.3.4 Заместитель

Проблема, которую решает паттерн Заместитель (Proxy) - выполнять какие-либо дополнительные действия при обращении клиента к серверному классу незаметно от клиента.

Мотивация использования Заместителя может быть самой различной. Например, Заместитель позволяет осуществлять контроль доступа к функциональности какого-то объекта, выполнять сохранение данных объекта или извлечение данных объекта, к которому обращается клиент, из базы данных или иного внешнего хранилища, либо –

осуществлять удаленный доступ к объекту-серверу с помощью каких-либо протоколов, известных только самому заместителю.

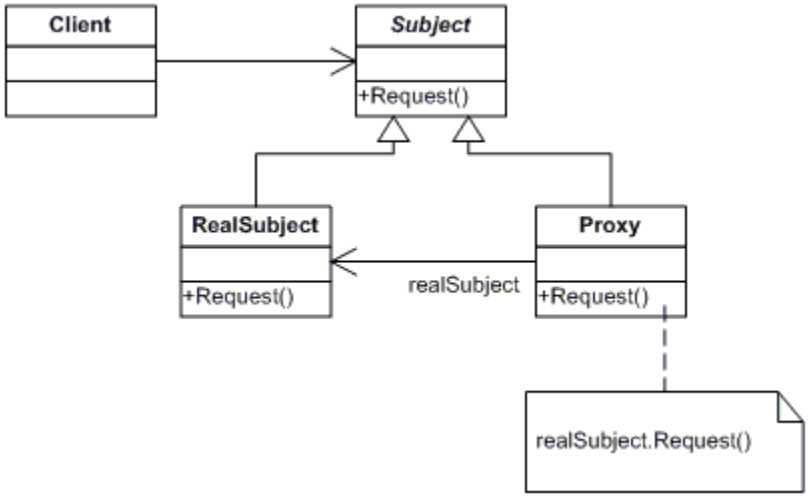


Рис. 4.5. Структура паттерна Заместитель

Структура паттерна представлена на рис. 4.5. Клиент Client обращается к серверному объекту с интерфейсом Subject с запросом Request() для выполнения каких-либо функций. Основные функции выполняются классом RealSubject, но наряду с ним реализуется его заместитель Proxy. Выполнение основной функции Request() Proxy делегирует объекту realSubject, а сам может выполнять дополнительные функции, в том числе перечисленные выше. Вероятнее всего, клиенту Client будет предоставлена ссылка именно на Proxy, а не на RealSubject.

Паттерн внешне похож на паттерн Адаптер, но предназначен совсем для другой цели – не согласование интерфейсов, а реализация под общим интерфейсом дополнительных функций, скрытых от клиента.

Как уже отмечалось, Proxy часто используется для работы с базами данных, а еще чаще – в распределенных системах, в обоих случаях, как правило, функциональность заместителей реализуется либо средствами библиотечных классов, либо классов, генерируемых по запросам пользователя. Например, для доступа к веб-сервису может генерироваться

Proxy класс, инкапсулирующий детали взаимодействия с веб-сервисом по протоколу SOAP.

4.2.4 Паттерны поведения

Паттерны поведения составляют наиболее многочисленную группу паттернов GoF. Их назначение в общем – инкапсулировать поведение класса или группы связанных классов. Так, Стратегия (она же Полиморфизм) – это инкапсуляция алгоритма, Шаблонный метод – инкапсуляция отдельных его шагов, Команда – инкапсуляция сценария, действия, транзакции, Итератор – инкапсуляция способа перебора составных объектов, а Наблюдатель – инкапсуляция реализация связи «один ко многим», реализуемой для слежения за изменяющимся состоянием какого-то объекта. Список паттернов поведения приведен в таблице 4.3.

Таблица 4.3. Список поведенческих паттернов проектирования GoF

№	Название паттерна	Перевод	Назначение паттерна
1	Chain of Responsibility	Цепочка обязанностей	Позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом объекты-получатели связываются в цепочку, а запрос передается по цепочке, пока какой-то объект его не обработает.
2	Command	Команда	Инкапсулирует запрос в виде объекта, обеспечивая параметризацию клиентов типом запроса, установление очередности запросов, протоколирование запросов и отмену выполнения операций.
3	Flyweight	Приспособленец	Использует принцип разделения для эффективной поддержки большого числа мелких объектов.
4	Interpreter	Интерпретатор	Для заданного языка определяет представление его грамматики на основе интерпретатора предложений языка, использующего это представление.

Окончание таблицы 4.3.

5	Iterator	Итератор	Дает возможность последовательно перебрать все элементы составного объекта, не раскрывая его внутреннего представления.
6	Mediator	Посредник	Определяет объект, в котором инкапсулировано знание о том, как взаимодействуют объекты из некоторого множества. Способствует уменьшению числа связей между объектами, позволяя им работать без явных ссылок друг на друга и независимо изменять схему взаимодействия
7	Memento	Хранитель	Дает возможность получить и сохранить во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить точно в таком же состоянии, не нарушая принципа инкапсуляции.
8	Observer	Наблюдатель	Специфицирует зависимость типа "один ко многим" между различными объектами, так что при изменении состояния одного объекта все зависящие от него получают извещение и автоматически обновляются.
9	State	Состояние	Позволяет выбранному объекту варьировать свое поведение при изменении внутреннего состояния. При этом создается впечатление, что изменился класс объекта.
10	Strategy	Стратегия	Определяет множество алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого. При этом можно изменять алгоритм независимо от клиента, который им пользуется.
11	Template Method	Шаблонный метод	Определяет структуру алгоритма, перераспределяя ответственность за некоторые его шаги на подклассы. При этом подклассы могут переопределять шаги алгоритма, не меняя его общей структуры.
12	Visitor	Посетитель	Позволяет определить новую операцию, не меняя описаний классов, у объектов которых она вызывается.

Далее рассмотрим некоторые паттерны поведения подробнее.

4.2.4.1 Стратегия

Проблему, которую решает паттерн Стратегия, легко понять, обратив внимание на синоним названия паттерна – Полиморфизм. Паттерн позволяет реализовывать различное поведение объектами с общим интерфейсом.

Мотивация использования Стратегии – спрятать от клиента детали реализации алгоритма за его интерфейсом. Это нужно везде, где алгоритм должен (или может) изменяться, а клиенту об этом лучше не знать. Причем в данном случае речь не обязательно идет о каком-то расчете, это может быть любой алгоритм с заданными входными и выходными параметрами.

Структура паттерна приведена на рис. 4.6. Клиенту Context предоставляется стратегия strategy с интерфейсом Strategy, позволяющим запускать алгоритм с интерфейсом AlgorithmInterface() – параметры запуска в данном случае не приведены. Конкретных стратегий может быть несколько, в данном случае показаны три.

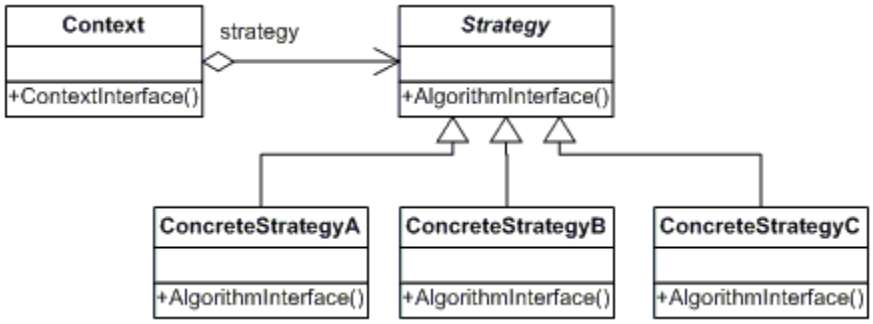


Рис. 4.6. Структура паттерна Стратегия

Стратегия, как и другой паттерн - Шаблонный метод, предназначена для инкапсуляции деталей алгоритма, но в отличие от Шаблонного метода, Стратегия скрывает весь алгоритм за специфицированным интерфейсом, а не отдельные его шаги.

Особенностью шаблона является то, что в нем за скобки вынесен существенный вопрос – кто, как и когда назначает конкретную стратегию клиенту Context. Это может реализовываться по-разному. К примеру, ссылка на стратегию может передаваться в конструкторе клиента (в этом случае алгоритм не изменяется в течение всего времени жизни объекта-клиента), либо устанавливается явно в каком-то методе клиента, либо может запрашиваться у какого-то фабричного объекта, в этом случае часто используются решения типа Абстрактной фабрики либо Фабричного метода, описанные далее.

Так или иначе, паттерн Стратегия должен быть хорошо знаком всем, кто изучал основы объектно-ориентированного подхода к проектированию и программированию и, в частности, понятие полиморфизма.

4.2.4.2 Состояние

Проблема, которую решает паттерн Состояние – объектно-ориентированная реализация модели конечного автомата, объекта с разными состояниями. Если несколько упростить, можно сказать, что Состояние – это версия Стратегии для модели конечного автомата.

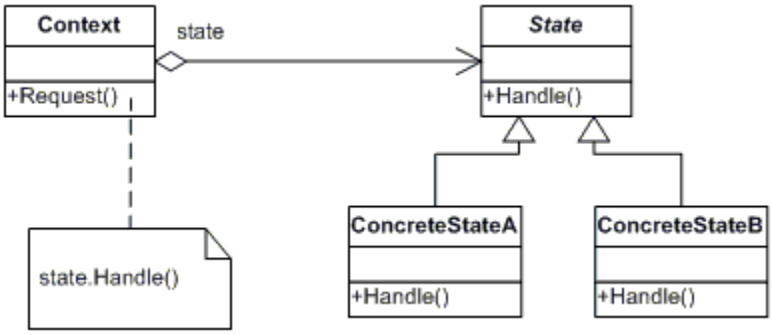


Рис. 4.7. Структура паттерна Стратегия

Даже структурно (рис. 4.7) данный шаблон похож на предыдущий, правда, в нем используется термин Состояние (State) – именно интерфейс Состояния с обработкой входящих событий, по-разному реализуемой в

разных конкретных состояниях объекта, дал название этому паттерну. Если представить себе программную реализацию конечно-автоматной модели, в традиционном виде это будут вложенные структуры типа switch-case и if-else, либо – интерпретация таблиц переходов. В данном шаблоне предполагается, что каждое состояние – это отдельный класс, по-своему реализующий интерфейс State. Отличие от Стратегии, помимо терминологии, заключается еще и в том, что разные состояния для выполнения перехода в другие состояния, скорее всего, вынуждены будут знать друг о друге, хотя это зависит от конкретной реализации шаблона.

4.2.4.3 Шаблонный метод

Шаблонный метод на первый взгляд решает проблему, похожую на ту, что решает Стратегия – инкапсуляция алгоритма. Однако, как уже было сказано выше при описании Стратегии, имеется важное отличие – нам нужно инкапсулировать отдельные шаги алгоритма, а не весь алгоритм.

Мотивация для использования данного паттерна похожа на мотивацию использования Стратегии – иметь разные реализации алгоритма с общим интерфейсом, но отличие в том, что сам алгоритм, его основная структура, будут общими для разных реализаций, отличаться будут только отдельные шаги (например, как описано в [9] – разные методы сортировки могут иметь общую структуру, но разные детали).

В итоге в данном паттерне общая часть алгоритма представлена в шаблонном методе TemplateMethod(), описанном в абстрактном классе AbstractClass, как показано на рис. 4.8. Отдельные изменчивые шаги алгоритма PrimitiveOperation1() и PrimitiveOperation2() (понятно, что в общем случае это может быть всего один шаг, могут быть и больше двух) оформлены в виде абстрактных методов (без реализации в классе AbstractClass), а уже конкретная их реализация выполняется в наследниках

ConcreteClass (на рисунке показан один наследник, но на практике их больше, иначе целесообразность паттерна будет под вопросом).

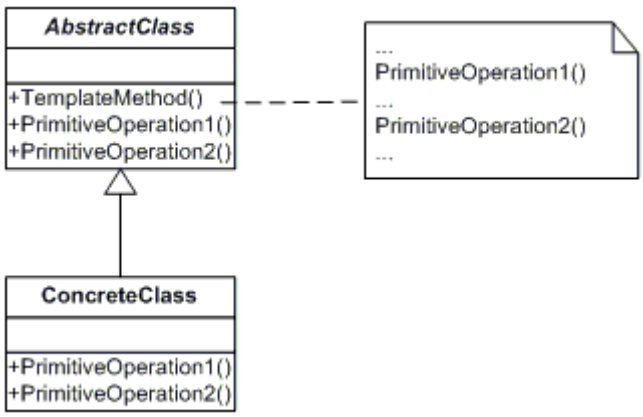


Рис. 4.8. Структура паттерна Шаблонный метод

Данный паттерн относится к паттернам уровня классов, а не объектов, как паттерн Стратегия, что еще раз указывает на их отличие. Но можно заметить, что этот паттерн, являясь результатом применения рефакторинга «Выделение шаблонного метода», часто на практике применяется вслед за выделением Стратегии, если программист видит, что у разных стратегий есть общая структура алгоритма, общие данные, есть что выделить в базовый (абстрактный) класс.

4.2.4.4 Команда

Назначение паттерна Команда - инкапсуляция запроса в виде объекта, что позволяет обеспечить параметризацию клиентов типом запроса, установить очередности запросов, выполнять протоколирование запросов и отмену выполнения операций.

Мотивация. Паттерн Команда – один из часто встречающихся паттернов поведения, который позволяет использовать абстракцию команды – то есть того, что можно выполнять (и отменять), не зная, с чем конкретно ты работаешь. Назначение Команды – спрятать от инициатора

команды обработчик какого-то сообщения, а также – обеспечить возможность единообразной обработки списка различных по сути, но одинаковых по интерфейсу команд.

Например, Команду можно использовать, чтобы : 1) вызывать одни и те же команды из разных мест пользовательского интерфейса; 2) не менять пользовательский интерфейс при изменении списка команд; 3) организовать отмену команд.

Кроме того, паттерн позволяет отделять инициацию команд от их реализации. Например, это помогает при программной реализации управления каким-то оборудованием, при реализации транзакций (в том числе проверка данных для транзакции, ее выполнение и откат), позволяет выполнять физическое и временное разделение связей между инициализацией и выполнением команд через работу со списком команд [8]. Как правило, этот паттерн используется во всех редакторах при реализации повтора и отмены последних действий (Redo и Undo пункта меню Edit). Да и при реализации паттерна работы с БД ActiveObject тоже может использоваться функциональность паттерна Команда.

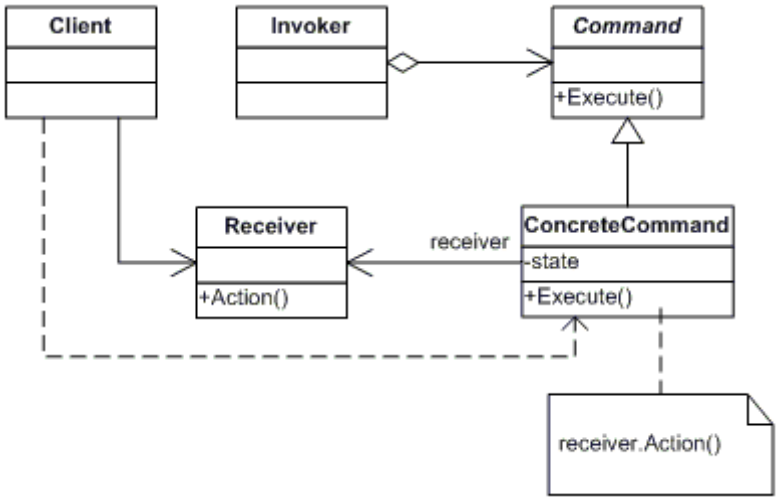


Рис. 4.9. Структура паттерна Команда

Структура паттерна показана на рис. 4.9. Участники паттерна: Command - интерфейс команды, ConcreteCommand – реализация команды, Receiver – исполнитель команды, Invoker – инициатор команды, запускает

команды на исполнение, работает со списком команд (инициатор не меняется при добавлении конкретных команд). Клиент – клиент, создающий новые команды и добавляющие их в список инициатора. Знает о получателе команд, но не вызывает его методы явно, предоставляя это инициатору. (Например, инициатором может быть обработчик меню программы).

Взаимодействие между участниками паттерна Команда показано на диаграмме последовательности на рис. 4.10.

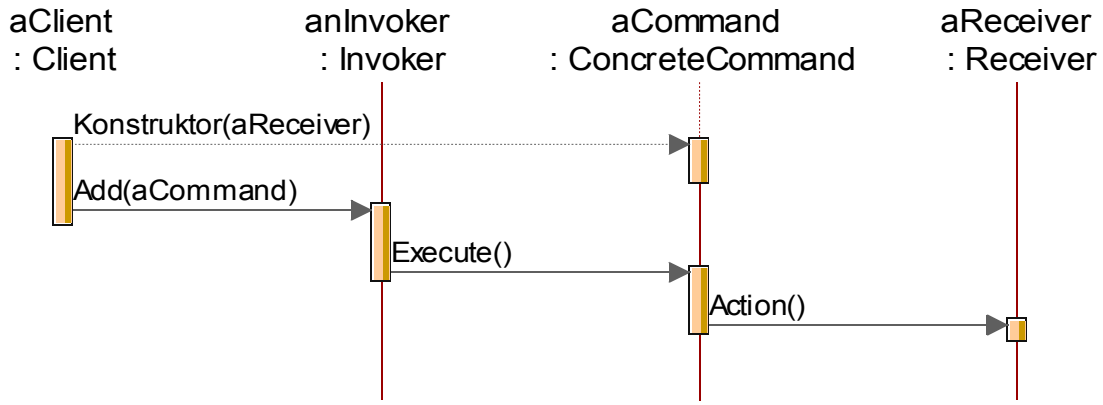


Рис. 4.10. Диаграмма последовательности для паттерна Команда

4.2.4.5 Итератор

Один из часто используемых паттернов поведения – Итератор (синоним Курсор), из-за повсеместного использования он встроен во многие языки и платформы программирования или стандартные библиотеки для них, в частности, в язык C++ и .NET Framework (и соответственно, в язык C#).

Проблема, для решения которой предназначен паттерн Итератор – инкапсуляция коллекции, или в более общей постановке – инкапсуляция деталей организации сложного объекта, элементы которого нужно перебирать поочередно. Мотивация для использования Итератора довольно очевидна – нам часто нужно иметь возможность работать с

составными частями чего-то целого, не вдаваясь в детали того, как это целое организовано, более того, часто нужно делать перебор по-разному, в зависимости от конкретных условий (например, обрабатывать не все элементы коллекции, а только выборочно, просматривать коллекцию в разном порядке и так далее).

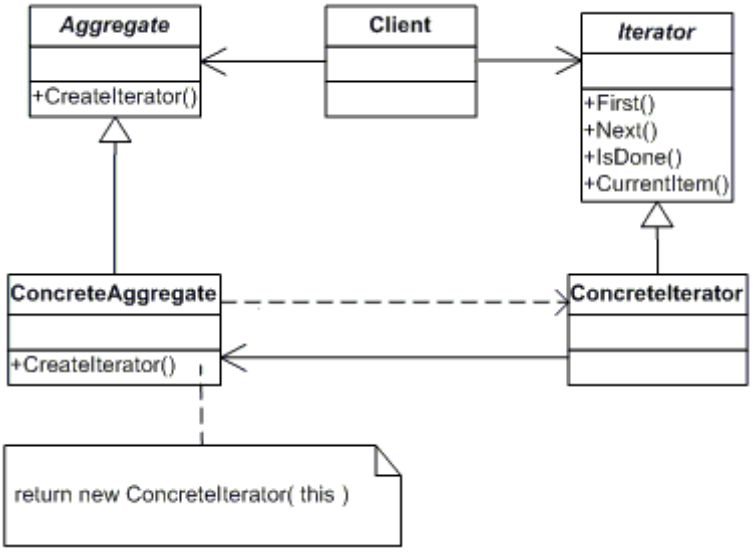


Рис. 4.10 – Структура паттерна Итератор

В структуре Итератора помимо традиционного для всех паттернов Клиента выделяются четыре основных участника, ключевым из которых, пожалуй, является интерфейс Iterator. Он позволяет перебирать необходимые элементы, связанные с классом-агрегатом, имеющим интерфейс Aggregate, с помощью методов First(), Next() и условия IsDone(), возвращающего true в случае окончания перебора. Также интерфейс итератора содержит метод CurrentItem(), возвращающий текущий элемент коллекции. Тип возвращаемого элемента в структуре паттерна не конкретизируется. Помимо интерфейсов имеются их реализации, при этом конкретный агрегат ConcreteAggregate содержит фабричный метод CreateIterator(), возвращающий ссылку на конкретный итератор ConcreteIterator, имеющий доступ к ConcreteAggregate и знающий, как осуществлять перебор связанной с ним коллекции.

Как уже отмечалось ранее, этот паттерн может быть связан с паттерном Фабричный метод, если метод CreateIterator() в свою очередь является абстрактным и реализуется конкретными классами, возвращающими разные итераторы для разных задач. Кроме того, в качестве агрегата ConcreteAggregate может выступать Компоновщик.

Интересно, что в силу реализации Итератора на уровня языка или платформы, сейчас уже редко моно встретить классическую реализацию в стиле, приведенном на рис. 4.10. Итераторы имеют самый разнообразный синтаксис. Так, в C++ вместо First() используется инициализация переменной итератора it значением begin() (либо метод коллекции, либо – начиная с C++11 - отдельная функция std::begin()), вместо IsDone() используется сравнение с end() (симметрично begin()), для перехода к следующему значению используется инкремент итератора it++, а вместо CurrentItem() – операция разыменования итератора *it. В .NET/C# используются стандартные параметризуемые интерфейсы IEnumerable<T> (аналог Aggregate) и IEnumerator<T> – собственно итератор. Последний больше похож на классический итератор, возвращающий по запросу Current очередное значение типа T (правда, это не метод, а свойство), вместо First() используется Reset(), а метод MoveNext() (аналог Next()) возвращает логическое значение, определяющее, не исчерпалась ли коллекция, так что IsDone() не нужен. В большинстве случаев программисту на C# не требуется явно реализовывать интерфейсы IEnumerator<T> и IEnumerable<T>, так как достаточно реализовать один метод-итератор, возвращающий IEnumerable<T> и содержащий оператор yield return, например :

```
List<Person> lst;
IEnumerable<Person> getPersons() {
        foreach(Person p in lst)
                yield return p;    }
```

Еще более своеобразной реализацией Итератора являются курсоры, используемые в скриптовых языках для серверов БД, например, в T-SQL компании Microsoft. Курсор можно трактовать как объект в памяти, хранящий результаты SQL-запроса SELECT и позволяющий перебирать их по одному с помощью операции FETCH. Фактически это тот же итератор, реализованный для не объектно-ориентированного языка, являющегося императивным дополнением декларативного языка SQL.

Аналогичная реализация курсора для работы с БД имеется и в библиотеке ADO.NET – это классы xDataReader, реализующие интерфейс IDataReader.

4.2.4.6 Наблюдатель

Паттерн наблюдатель (Observer), известный также как «Опубликовать-Подписаться» (Publish-Subscribe), как и Итератор является практически стандартным элементом многих языков или, по крайней мере, программных платформ.

Паттерн специфицирует зависимость типа "один ко многим" между различными объектами, так что при изменении состояния одного объекта все зависящие от него получают извещение и автоматически обновляются.

Мотивация. Паттерн Наблюдатель иллюстрирует решение типичной проблемы обратного вызова – как обеспечить вызов серверным классом методов своих клиентов, не привязывая сервер к типам клиентов (сервер не должен зависеть от клиентов). Эта проблема часто возникает при отделении слоя предметной области (или просто – слоя логики приложения) от пользовательского интерфейса (что также является задачей соответствующего рефакторинга). Создавая в слое логики приложения «дублирующиеся данные» (опять же прием рефакторинга), мы должны обеспечивать их синхронизацию, в чем помогает данный паттерн.

В более простом случае проблема возникает, например, при необходимости управлять слоем пользовательского интерфейса из «недр» программы. Допустим, мы хотим сделать пользовательский интерфейс настолько малофункциональным, чтобы он только передавал в наш контроллер прецедента информацию о происходящих событиях, а сам никаких решений не принимал («пассивный» клиент). Тогда, например, при необходимости выполнения проверки вводимых данных уже в слое предметной области и, главное, блокировки ввода по инициативе самого контроллера, нам потребуется как-то вызывать из контроллера методы формы, что нарушает принцип шаблона Слои. Выход состоит в использовании механизма обратного вызова, предлагаемого паттерном Наблюдатель, чтобы «подписать» форму в качестве наблюдателя за сообщениями предметной области. Контроллер будет знать о существовании абстрактных наблюдателей за ним, рассылать им сообщения, при этом не зная о реальных классах, реализующих этот интерфейс.

На практике, однако, более полно возможности этого паттерна раскрываются при необходимости именно синхронизации данных, особенно когда наблюдателей несколько.

Структура. Классический вид паттерна Наблюдатель демонстрируется на диаграмме, приведенной на рис. 4.11. Участники паттерна: Observer – интерфейс наблюдателей, Subject – абстрактный класс (иногда – интерфейс) наблюдаемого объекта (в нашем случае – контроллера или класса предметной области), ConcreteObserver – конкретный наблюдатель (в нашем случае – Форма), ConcreteSubject – конкретный наблюдаемый объект (в нашем случае - контроллер или класс предметной области). Ключевым решением, конечно, является введение интерфейса Observer.

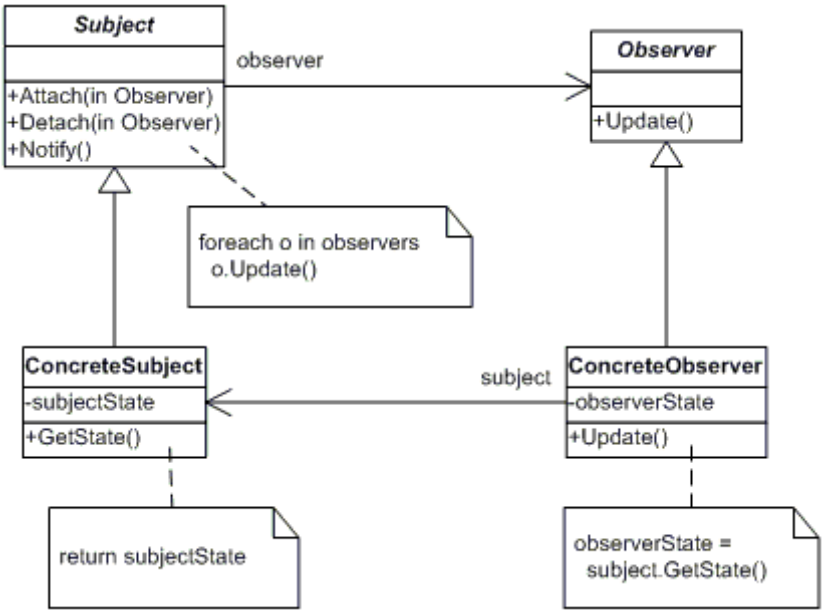


Рис. 4.11 – Структура паттерна Наблюдатель

Так как класс-сервер (ConcreteSubject) не должен знать о типах наблюдающих за ним объектов, логично ввести для них общий интерфейс. Другим важным элементом паттерна является абстрактный класс Subject, который содержит весь основной механизм работы с наблюдателями – их список observers, методы редактирования списка (Attach(), Detach()), а также сосбственно метод оповещения наблюдателей Notify().

На рис.4.12 показана диаграмма последовательности паттерна :

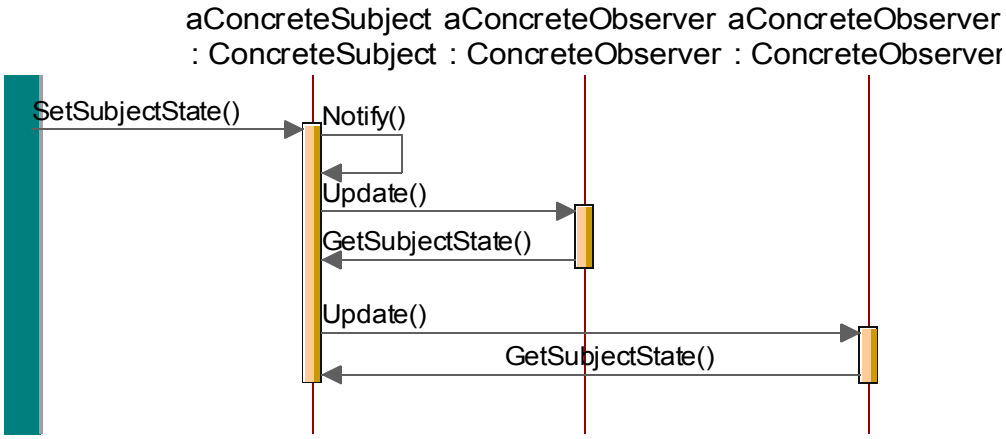


Рис. 4.12 – Диаграмма последовательности для паттерна Наблюдатель

После изменения состояния наблюдаемого объекта он вызывает свой метод Notify(), который он наследует от родителя Subject, в этом методе каждому наблюдателю рассылается оповещение о необходимости обновиться (повторно запросить состояние наблюдаемого объекта и что-то затем сделать).

Из особенностей реализации паттерна необходимо выделить две его разновидности - в классической, показанной на рис. 4.11-4.12, используется т.н. механизм «вытягивания» наблюдателя, когда ConcreteObserver() сам запрашивает новое состояние у наблюдаемого объекта ConcreteSubject(), получив уведомление Update(). Вторая разновидность использует механизм «толкания» наблюдателя, для этого в методе Update() можно предусмотреть параметр вызова, содержащий нужные для обновления данные. В этом случае связь между ConcreteObserver и ConcreteSubject() может отсутствовать, чем потенциально достигается еще большая гибкость и независимость кода.

Паттерн Наблюдатель применяется во многих библиотеках графического интерфейса пользователя (GUI, например в тех же Windows Forms в .NET, в Qt), в объектных каркасах, например, он реализован в платформе Java2ME (для мобильных устройств) и вообще в различных событийно управляемых интерфейсах.

В языке C# для реализации Наблюдателя используется специальный механизм языка, использующий события Events и делегаты delegates, представляющие собой описания сигнатур методов (аналог указателей на функции в C), необходимых для обработки событий :

```
delegate void MyDelegate(Object obj, EventArgs arg);
public class Sample {
   public event MyDelegate MyEvent;
   MyEvent(this, args); // Где-то в теле метода класса Sample… }
Sample mySample;   mySample.MyEvent += someDelegate; // где-то в программе
```

4.2.5 Паттерны создания

Паттерны создания, как уже упоминалось, нужны для инкапсуляции деталей создания объектов. Список паттернов создания приведен в таблице 4.4. Наиболее часто используются 3 паттерна: Одиночка, Фабричный метод и Абстрактная фабрика, описанные далее.

Таблица 4.4. Список паттернов создания GoF

№	Название паттерна	Перевод	Назначение паттерна
1	Abstract Factory	Абстрактная фабрика	Предоставляет интерфейс для создания множества связанных между собой или независимых объектов, конкретные классы которых неизвестны.
2	Builder	Строитель	Отделяет создание сложного объекта от его представления, позволяя использовать один и тот же процесс разработки для создания различных представлений.
3	Factory Method	Фабричный метод	Определяет интерфейс для разработки объектов, при этом объекты данного класса могут быть созданы его подклассами.
4	Prototype	Прототип	Описывает виды создаваемых объектов с помощью прототипа, что позволяет создавать новые объекты путем копирования этого прототипа.
5	Singleton	Одиночка	Для выбранного класса обеспечивает выполнение требования единственности экземпляра и предоставления к нему полного доступа.

4.2.5.1 Одиночка

Проблема, которую решает данный паттерн, состоит в необходимости иметь в какой-то системе строго один экземпляр какого-то объекта. Например, это может быть единственный экземпляр кэша объектов-сущностей предметной области, единственный объект для оступа к базе данных или единственный объект, хранящий конфигурацию программы.

Структура паттерна (рис. 4.13) состоит всего из одного класса Singleton, который использует статическое поле instance типа Singleton и метод Instance(), возвращающий ссылку на это поле.

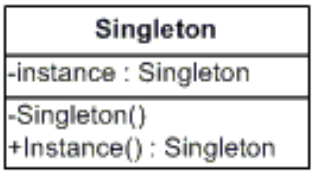


Рис. 4.13. Структура паттерна Одиночка

4.2.5.2 Фабричный метод

Паттерн Фабричный метод позволяет решить, в частности, следующую проблему. Если какой-то объект класса X использует объекты с определенным интерфейсом I и ему периодически нужно их создавать, но он не знает конкретные типы создаваемых объектов, поскольку они могут меняться. В частности, тип создаваемых объектов может зависеть от хода выполнения метода класса X, нуждающегося в новых объектах с интерфейсом I, либо он зависит от общего контекста, в котором работает X. В данном случае возможное решение может состоять в создании специального абстрактного фабричного метода в классе X, который будет возвращать объект с интерфейсом I по запросу самого X, но конкретная реализация этого метода будет выполняться только в наследнике X, который будет знать о типе нужного класса с интерфейсом I.

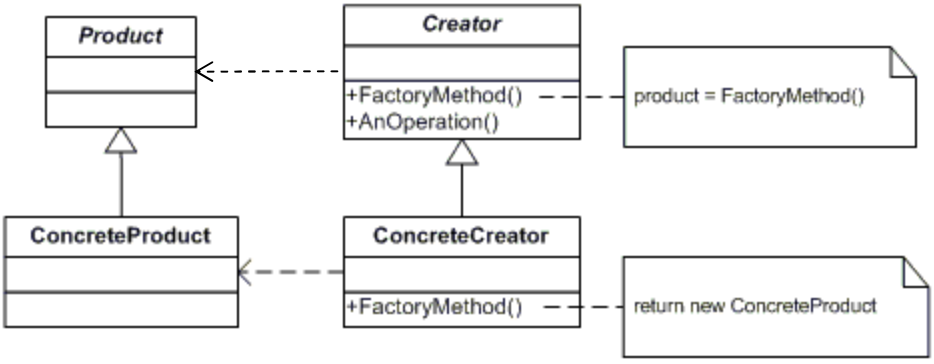


Рис. 4.14. Структура паттерна Фабричный метод

Структура паттерна показана на рисунке 4.14. Роль класса X играет класс Creator с фабричным методом FactoryMethod(). Понятие «фабричный» означает, что метод создает объект с заданным типом (в

данном случае – с типом интерфейса Product). Обычно, говоря о фабрике классов, имеют в виду отдельный класс, создающий объекты одного или нескольких типов, в данном случае речь идет только о фабричном методе обычного класса Creator, не являющегося фабрикой. Объект product используется самим классом Creator, например, в методе AnOperation(). Реализуется метод FactoryMethod() в наследнике ConcreteCreator, который знает о конкретном создаваемом продукте ConcreteProduct. Данный паттерн является паттерном уровня классов.

Паттерн Фабричный метод может использоваться совместно с другими, например, с паттернами Компоновщик и Итератор (продуктом может являться возвращаемый итератор).

4.2.5.3 Абстрактная фабрика

В предыдущем пункте упоминалось понятие фабрики классов. В общем случае фабрика классов создает объекты («продукты») одного или нескольких заданных типов. Представим себе ситуацию, когда в нашей программе имеются клиенты (классы), нуждающиеся в ряде объектов (продуктов), при этом конкретные типы этих объектов классам-клиентам должны быть неизвестны, а создавать их нужно динамически в ходе работы, то есть нельзя один раз проинициализировать клиентов нужными объектами. Эта ситуация далеко не надуманная, а вполне реальная. Допустим, те или иные стратегии зависят от настроек, которые выбирает пользователь в процессе работы программы, причем заново создать объект класса-клиента, назначив ему новую стратегию, нет возможности. Например, объект клиентского класса нуждается в новой стратегии в ходе выполнения какого-то более крупного алгоритма, а тип стратегии зависит от внешних обстоятельств, например, от тех же настроек, выбранных пользователем. Конечно, где-то в программе может существовать некий

объект-контроллер, управляющий работой такого объекта-клиента, нуждающегося в стратегиях, но тогда ему нужно постоянно отслеживать необходимость создания стратегий, либо он должен уведомляться о ней через механизм Наблюдателей, что не всегда удобно. Кроме того, этот контроллер должен знать и обо всех возможных стратегиях, что делает его реализацию иногда слишком сложной и привязанной к контексту.

С другой стороны, даже если описанная ситуация с динамическим созданием продуктов неизвестного типа не возникает, в достаточно больших программах процесс создания объектов-продуктов просто для удобства выделяется в отдельный класс-фабрику. И если в этом случае нам потребуется выделить фрагмент нашего приложения, использующий эту фабрику, в отдельную библиотеку для повторного использования, именно зависимость от фабрики может помешать нам это сделать.

Выход в описанных случаях – использование паттерна Абстрактная фабрика (рис. 4.14). В структуре паттерна помимо клиента Client, нуждающегося в продуктах AbstractProductA и AbstractProductB, выделяется интерфейс абстрактной фабрики AbstractFactory (ключевой участник), в котором описаны методы создания абстрактных продуктов CreateProductA() и CreateProductB(), а также для примера две ее реализации ConcreteFactory1 и ConcreteFactory2, каждая возвращающая свой набор конкретных продуктов A и B с номерами 1 и 2.

В результате, если мы хотим выделить подсистему, содержащую класс Client и интерфейсы нужных ему продуктов, в отдельную автономную библиотеку, мы можем это сделать, снабдив их интерфейсом абстрактной фабрики. Этот паттерн очень важен для грамотной декомпозиции проектов по подсистемам и упаковки подсистем в автономные библиотеки.

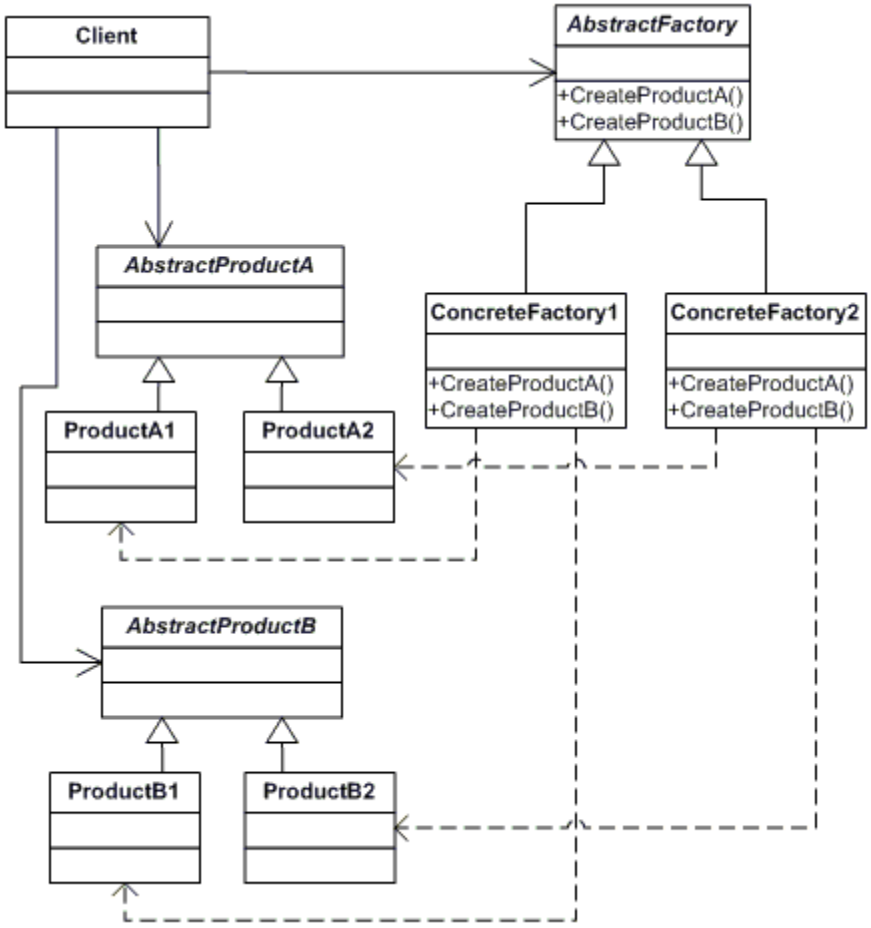


Рис. 4.14. Структура паттерна Абстрактная фабрика

Помимо [17] многочисленные примеры использования паттернов GoF приведены в [8], также для неформального изучения паттернов на примерах можно порекомендовать книгу [19].

4.2.6 Рефакторинг и паттерны

Рефакторинг, о котором шла речь в разделе 3, помимо уборки и устранения неприятных запахов в коде определяет путь эволюционного проектирования, при котором можно из совершенно необъектного проекта получить объектный путем простых несложных мелких шагов. Как и TDD в целом, рефакторинг не является «серебряной пулей» и чаще всего должен сочетаться с другими практиками, например, быстрым

проектированием, моделированием и пр. Интересна в этом смысле беседа двух разработчиков во время эпизода парного программирования, приведенная, например, в [9].

С другой стороны, рефакторинг часто приводит нас к тем решениям, которые можно было бы предложить сразу, имея мы соответствующий опыт проектирования. В том числе, путем рефакторинга мы приходим к решениям, предлагаемым паттернами проектирования. Таким образом, с одной стороны, выполняя постепенно рефакторинг, мы можем получить решения, которые потом можно улучшить, доработать, используя паттерны. С другой стороны, мы можем, выполняя рефакторинг, сразу оформлять какие-то более крупные модификации кода, применяя нужные паттерны проектирования. Подробнее это описано, например, в книге [20].

ЗАКЛЮЧЕНИЕ

В учебном пособии рассмотрен адаптивный (или гибкий) подход к разработке программного обеспечения, получивший достаточно широкое распространение с начала 2000 годов благодаря эффективному итеративному планированию, концентрации на качестве кода, которое достигается его переработкой и исчерпывающим модульным тестированием, а также готовностью и проекта, и коллектива разработчиков к постоянным изменениям требований, что достигается в том числе использованием паттернов проектирования, инкапсулирующих изменяемые фрагменты кода.

Многие важные вопросы, относящиеся к адаптивным технологиям, к сожалению, вышли за сжатые рамки данного пособия, в частности, шаблоны создания корпоративных приложений, ориентированные на отдельные их подсистемы (шаблоны интерфейса пользователя, предметной области, шаблоны работы с БД) подробно описаны в книге [21]. Заинтересованному читателю мы рекомендуем все источники [1-21].

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Одинцов И.О. Профессиональное программирование. Системный поход.–2-е изд. перераб.– СПб.: БХВ-Петербург, 2004.–624с.:ил.
2. Благодатских В.А. и др. Стандартизация программных средств: Учеб. пособие / В.А. Благодатских, В.А. Волнин, К.Ф. Поскакалов; Под. ред. О.С. Разумова. –М.:Финансы и статистика, 2005.–288 с.:ил.
3. Ларман К. Применение UML и шаблонов проектирования. 2-е изд.– М.: Вильямс. 2004. –624с: ил.
4. Фаулер М., Скотт К. UML. Основы, 3-е изд. – СПб.:Символ-плюс, 2005. – 192 с: ил.
5. Бек К. Экстремальное программирование - № 01-02, 2000 | Открытые системы [Электронный ресурс] Режим доступа : http://www.osp.ru/os/2000/01-02/178193/
6. Ривз Д. Как проектировать ПО ? [Электронный ресурс] Режим доступа : http://old.computerra.ru/offline/2005/589/38835/
7. Макконнелл С. Совершенный код. Практическое руководство по разработке программного обеспечения.–СПб.: Питер, 2005.–896 с.
8. Мартин Р., Мартин М. Принципы, паттерны и методики гибкой разработки на языке C#. – СПб: Символ-Плюс, 2011. – 768 с.:ил.
9. Бек К., Фаулер М. Экстремальное программирование: планирование. Библиотека программиста.–СПб: Питер, 2003. – 144 с.:ил.
10. Бек К. Экстремальное программирование: разработка через тестирование. Библиотека программиста.– СПб: Питер, 2003. – 224 с.
11. Эккель Б., Эллисон Ч. Философия C++. Практическое программирование.– СПб: Питер, 2004. – 608 с.
12. Томас Д., Хант Э. Учимся любить юнит-тесты. [Электронный ресурс] Режим доступа : http://xp.1024.info/Articles/LoveUT.html
13. NUnit [Электронный ресурс] Режим доступа: http://www.nunit.org/

14. Эндрю Хант, Дэвид Томас: Pragmatic Unit Testing in C# with NUnit (Прагматичное тестирование на C# с помощью NUnit). The Pragmatic Bookshelf.-Raleigh, 2004.
15. Фаулер М. Рефакторинг: улучшение существующего кода. СПб: Символ-Плюс, 2004 –430 с.
16. Fowler M. Catalog of Refactorings [Электронный ресурс] Режим доступа: http://refactoring.com/catalog/
17. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер. 2007. 366 с.
18. .NET Design Patterns in C# and VB.NET – Gang of Four (GOF) – doFactory.com [Электронный ресурс] Режим доступа: http://dofactory.com/net/design-patterns
19. Фримен Э., Фримен Э., Сьерра К., Бейтс Б. Паттерны проектирования. – СПб.: Питер, 2011. –656 с.:ил.
20. Кириевски Д. Рефакторинг с использованием шаблонов.-М.:Вильямс, 2008. – 400 с.: ил.
21. Фаулер М. и др. Шаблоны корпоративных приложений. – М.: Вильямс, 2010. – 544 с.: ил.

Учебное издание

Андрей Евгеньевич **Андреев**
Семен Игоревич **Кирносенко**

АДАПТИВНЫЕ ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Выпускающий редактор *Л. П. Кузнецова*

Темплан 2015 г. (учебники и учебные пособия). Поз. № 43.
Подписано в печать 21.12.2015. Формат 60x84 1/16. Бумага газетная.
Гарнитура Times. Печать офсетная. Усл. печ. л. 5,58. Уч.-изд. л. 4,17.
Тираж 100 экз. Заказ 871

Волгоградский государственный технический университет.
400005, г. Волгоград, просп. им. В. И. Ленина, 28, корп. 1.

Отпечатано в типографии ИУНЛ ВолгГТУ.
400005, г. Волгоград, просп. им. В. И. Ленина, 28, корп. 7.