“Теория без практики мертва, а практика без теории слепа.”

“Теория это когда все понятно, но ничего не работает,
практика - когда все работает, но ничего не понятно.
Но бывают моменты, когда практика соединяется с теорией -
это когда ничего не работает и ничего не понятно...”

Тема 1
Краткое содержание и цели предмета. Различные процессы разработки программ (каскадная модель, итеративная разработка, адаптивный процесс).

План лекции :

1. Цели учебной дисциплины согласно ГОС.
2. Неформальное определение целей дисциплины. Для кого и для чего предназначена.
3. Содержание дисциплины (основные разделы курса).
4. Рекомендуемая (очень рекомендуемая !) литература и источники.
5. Виды учебных поручений.
6. Роль процессов разработки. “Треугольник успеха”.
7. Традиционный каскадный процесс. Преимущества и недостатки.
8. Итеративный процесс (в частности, RUP). Преимущества и недостатки.
9. Понятие об адаптивной разработке. (Особенности. Преимущества и недостатки).

1. Цели учебной дисциплины.

Целью данной дисциплины должно являться изучение технологий разработки программ в различных смыслах. То есть вообще здесь имеются в виду технологические процессы создания программ, фундаментальные методологии или парадигмы программирования (например, объектно-ориентированные технологии), более локальные технологии выполнения определенных этапов создания программ (методики проектирования, анализа требования, кодирования, тестирования и пр.), технологии создания определеных компонентов программ (технологии работы с данными, технологии создания интерфейса и пр.), специфичные технологии (для данной платформы, данного протокола, данной системы и пр.) Как видите, это очень обширная тема. В образовательных стандартах в контексте этой дисциплины упоминаются : “Задача проектирования программных систем; организация процесса проектирования программного обеспечения (ПО); использование декомпозиции и абстракции при проектировании ПО; специфики процедур и данных; декомпозиция системы; методы проектирования структуры ПО; методология объектно-ориентированного программирования; технологические средства разработки программного обеспечения: инструментальная среда разработки, средства поддержки проекта, отладчики; методы отладки и тестирования программ; документирование и оценка качества программных продуктов; методы защиты программ и данных; проектирование интерфейса с пользователем; структуры диалога; поддержка пользователя; многооконные интерфейсы; примеры реализации интерфейсов с пользователем с использованием графических пакетов. “ Особенно загадочно звучит последний пункт.

Конечно, мы не сможем охватить все виды и аспекты технологий. Даже краткое знакомство с ними займет очень много времени. Цель курса в общем получить представление о некоторых важных современных технологиях и методологиях разработки ПО более универсального и фундаментального характера.

2. Неформальное определение целей курса. Для кого и для чего.

Ну, прежде всего курс предназначен для программистов и интересующихся программированием. Это вполне естественно. Это могут быть и будущие профессиональные программисты, причем специализирующиеся в самых разных областях (от создания объектных библиотек для работы с графикой до программистов 1С), и системщики, и даже специалисты по проектированию аппаратных средств. Последние ведь тоже занимаются программированием, кроме того, сейчас проектирование АС часто выполняется на HDL. Кроме того, задачи проектирования ПО и АС при всей непохожести имеют и общие черты. Все же наиболее полезен курс будет, вероятно, будущим профессиональным программистам, использующим объектный подход.

Для них, да и для всех остальных категорий цели этого курса неформально можно определить следующим образом : прояснить назначение некоторых модных новых (и уже не очень новых) терминов и сокращений в области создания ПО, рекомендовать определенную литературу, дать некоторые полезные практические навыки (например, в области тестирования ПО и его улучшения и др.) и навести на некоторые новые мысли по поводу типичных фраз для студентов (да и не только), занимающихся программированием, например :

“Да, конечно, моя программа является объектной, она ведь написана на Delphi ...”

“Программа готова, нужно только ее отладить, а то она часто вылетает, зависает и не всегда правильно считает.”

“Странно, здесь программа не работает, а дома работает почему-то...”

“Я, к сожалению, не могу показать работу программы здесь, так как она сетевая, а здесь нет второго компьютера / сервера / нужного устройства ...”

“Чтобы добавить эту возможность, мне придется переписывать всю программу...”

“Веб интерфейс ? Не, это мне всю программу переделывать !”

“Моя программа работает только с MS SQL Server !”

“Без ТЗ я к работе приступать не буду !”

“Пока программа не работает, но завтра к утру ...”

“Я мало что еще пока реализовал, но у меня ведь еще неделя”.

“Да, это я еще не доделал, зато смотрите, что моя программа может делать !”

“Программа описана в приложении, там ТЗ, руководство программиста, а в записке описана структура базы данных...”

“Документация к программе есть, там много комментариев...”

3. Содержание курса (основные разделы)

2) Процессы разработки, прежде всего итеративные (UP). Быстрая разработка. Экстремальное программирование (XP) и его практики (планирование, разработка через тестирование, рефакторинг, участие заказчика, парное программирование…)
3) Объектные технологии (анализ, проектирование, программирование).
4) Роль UML. Использование UML.
5) Шаблоны проектирования. (GoF, Ларман)
6) Построение модульных, многослойных приложений. Архитектурное проектирование.
7) Проектирование пользовательского интерфейса и взаимодействие с БД.
8) Место и роль инструментальных средств : CASE, RAD, UML Tools, xUnit, RAD. Командная разработка.
9) Технологии распределенных приложений и др.

4. Литература

Общая теория и классификация технологий программирования

1. Камаев В.А., Костерин В.В. Технологии программирования
2. Сикоров. Профессиональное программирование. Системный подход

Экстремальное программирование и быстрая разработка.

3. Р.Мартин, Р.Косс, Д. Ньюкирк Быстрая разработка программ
4. К. Бек Экстремальное программирование
5. К. Бек Разработка через тестирование
6. К. Бек и М. Фаулер Экстремальное планирование

Язык UML

7. М. Фаулер «UML в кратком изложении» 3-е издание
8. Г.Буч, И.Якобсон, Д.Румбах «UML. Руководство пользователя»

Паттерны проектирования и их применение.
Объектное проектирование.

8. Р. Хелм, Джонсон, Дж. Влиссидес, Э. Гома «Паттерны проектирования» (GoF)
9. М. Фаулер "Архитектура корпоративных приложений
10. К. Ларман "Применение UML и шаблонов проектирования. Унифицированный процесс" 2-е изд.

Унифицированный процесс

12. К. Скотт "Концепции RUP"
13. Буч, Якобсон, Рэмбо "RUP"

Рефакторинг

14. М. Фаулер "Рефакторинг или Улучшение существующего кода"

Общие рекомендации по программированию и кодированию

15. Мак-Конел С. "Совершенный код"

Ссылки на ресурсы в Internet

1. Ресурс, посвященный XP www.xprogramming.com
2. Российский ресурс по XP : www.xprogramming.ru
3. Страница М. Фаулера www.martinfowler.com
4. Страница К. Ларманa www.craiglarman.com
5. Страница фирмы Object Mentor www.objectmentor.com

5. Виды учебных поручений

Лабораторные работы – 8 штук (примерный план). Семестровое задание.

6. Роль процессов разработки. "Треугольник успеха" (?)

Как же можно определить основные составляющие процесса разработки ? Это можно сделать по-разному. Например, Терри Кватрани в своей книжке про использование UML и Rational Rose говорит от т.н. «треугольнике успеха» :

- процесс;
- нотация;
- инструменты.

Под **процессом** подразумевается, прежде всего, **процесс разработки** («водопад», UP, RUP, XP).

7. Различные процессы разработки

Поговорим немного о **процессе**. Этот разговор предваряет чуть более подробную беседу об XP. Итак, я упомянул выше три модели процесса разработки ПО (или жизненного цикла ПО): «водопад», RUP, XP. На самом деле процесс включает в себя больше, чем только понятие жизненного цикла, но начнем с него.

Итак, **водопад** - это традиционный процесс разработки, в котором четко выделяются последовательные этапы:

- общая постановка задачи, анализ требований;
- составление технического задания;
- составление технического проекта;
- реализация проекта;
- отладка проекта;
- поставка программного продукта;
- сопровождение продукта.

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 хорошо соответствует документации ЕСПД, но уже в течении многих лет показывает свою несостоятельность на практике. Она очень неповоротлива, и главные ее недостатки :

- отсутствие гибкости (все изменения вносятся в уже готовую программу на этапе сопровождения, либо – составляется новый проект и все сначала);
- очень трудно полностью сформулировать все требования ДО начала собственно кодирования и до того, как заказчик познакомится с 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 !);
- процесс требует значительного времени.

Все это привело в конечном итоге к появлению унифицированного процесса разработки, и прежде всего – в виде унифицированного процесса Rational (RUP). Это многосложный процесс, включающий много концепций, но в плане жизненного цикла он отличается чем :

1) С точки зрения этапов проектирования он в общем аналогичен «водопаду» :

- задумка (видение);
- анализ требований;
- фаза развития;
- фаза конструирования;
- фаза поставки.

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, анализом и фазой развития.

2) Ключевой идеей процесса является его итеративность – каждая фаза, начиная с развития, включает несколько итераций, на каждой из которых выполняется свой кусочек анализа, проектирования, реализации и тестирования готового продукта (вернее его части).

Задачи итерации определяются прецедентами использования (use cases), которые отражают функциональные требования к продукту с точки зрения пользователей. Так постепенно система наращивает функциональность до полной. 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 решения уже приняты (до 70-80%) и на этапе конструирования в рамках этих решений система доводится до конца. Большое внимание уделяется этапу задумки и анализу требований, и каждая итерация (особенно на этапе развития) предваряется серьезным проектированием, в частности, диаграммами UML (чаще всего используются диаграммы классов и последовательностей (кооперации)). В результате перед написанием собственно кода итерации существует фактически мини- техпроект с указанием всех основных модулей (классов), а также – с расписанным их поведением в виде диаграмм кооперации. Кстати, оговаривается, что длительность одной итерации небольшая - от 2 до 6 недель.

3) 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.

Основные идеи RUP изложены в многочисленных публикациях, из наиболее доступных можно назвать Лармана /3/ (который оговаривается, что использует свой собственный аналогичный RUPу процесс, но более абстрактный, и называет его UP) .
Собственно RUP разработан сотрудником Rational Software Филиппом Крачтеном и вдохновлен (и поддерживается) отцами-основателями Rational Бучем, Якобсоном и Рэмбо (Румбахом) – т.н. «три амигос» (они же авторы UML).

Процесс хорош, в особенности его принцип итеративности, но обладает рядом недостатков :

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

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 оставлят на потом 30%). В реальных ситуациях и 70% сразу получить трудно !

Дальнейшим развитием (или специфической реализацией) данного подхода является модель жизненного цикла в экстремальном программировании (XP).

Эта модель предполагает :

- еще более жесткие ограничения на длительность одной итерации (2 недели);
- резкое сокращение времени, затрачиваемого на анализ и проектирование (в том числе – уменьшение кол-ва разрабатываемых диаграмм UML, уменьшение подробности анализа задач – вместо прецедентов рассматриваются коротки е (в несколько предложений) «пользовательские истории»;
- 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 ! Подробный план составляется на ближайшие 1-2 итерации ! Существенное значение имеет планирование от достигнутого – то есть по окончании итерации оценивается количество решенных задач за итерацию, и составляется план на следующую (1-2), исходя из существующей скорости работы.

Эта модель вызвана к жизни необходимостью экономить средства заказчиков и свое время. То есть взаимодействие с заказчиком осуществляется гораздо более плотное и дальнейшее сотрудничество зависит от результатов очередной итерации (или цикла), 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 (знакомо, не правда ли ? ☺) этот процесс, пожалуй, наиболее близок к жизни. Я не хочу сказать, что RUP (UP) плох, а XP – хорошо, тем более , я недостаточно знаю UP.

Конечно, для достижения положительного результата в таких условиях XP предлагает наряду с изложенными ряд принципов, которые мы рассмотрим ниже.

Тема 2
Введение в быструю разработку (agile development). Принципы и практики экстремального программирования (XP).

План лекции :
1. Принципы быстрой разработки (продолжение)
2. Манифест альянса быстрой разработки. Его смысл.
3. Принципы и практики экстремального программирования.
4. Переход к XP. Использование практик XP. XP и другие методики.

> *В моем представлении, название «экстремальное программирование» было выбрано по аналогии с «экстремальными видами спорта»: воздушным серфингом, парашютированием с небоскребов и тому подобным. Другими словами - опасное, стоящее на грани, искусное. Проблема в том, что я представляю картину, где парень с ноутбуком программирует на Delphi, совершая прыжок в воду с пятидесятиметровой скалы!*
>
> *- И чем это отличается от обычного программирования*
>
> Из форума обсуждения экстремального программирования в конференции borland.public.delphi.non-technical)

2. Основные концепции XP

Манифест быстрой разработки - идеология XP.

Основная идеология XP, как и других методов быстрой разработки (т.н. agile development) , пожалуй, отражается в манифесте альянса быстрой разработки (agile alliance). Вот как звучит этот манифест, который был подписан в конце 2001 года рядом признанных специалистов в области объектных технологий и теории проектирования / 1 / :

«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.»

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

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 получает именно сейчас.

Если рассматривать данный манифест, то он декларирует приоритет человеческого фактора, программного кода, готовности к изменениям и общения как внутри команды так и с заказчиком над всеми остальными компонентами проекта.

3 Основные принципы и практики XP.

Основные принципы и используемые 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. (“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.

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

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».

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.

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.

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.»

Роберт Мартин / 1 / 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» … »

(Замечу, что последнее требование соответсвует требованию к 40-часовой рабочей неделе и отказу от сверхурочной работы).

От себя добавлю, 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 (принцип «Это нам никогда не понадобится !»);
- для ООП-разработчиков советуют следовать базовым правилам объектного проектирования (см. далее), а для всех – просто **разумным** правилам сохранения простоты (не допускать дублирования кода, все делать максимально простым образом, использовать выразительные комментарии и имена переменных и функций и т.д.)

Многие авторы прямо указывают, что принципы и практику XP следует вводить **постепенно**. В этом плане интересны, например, замечания тех же авторов с сайта www.xprogramming.ru . Они представляют собой команду московских разработчиков ПО, которая пытается применить на практике принципы XP.

С моей точки зрения, помимо собственно жизненного цикла проекта из XP можно непосредственно сразу (или очень быстро) можно начать применять следующие практики:

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

XP является гибким и простым для понимания процессом, 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 и объектным технологиям. Однако он не является панацеей для всех видов проектов, в данном случае многое определяется типом самого проекта.

В частности, это хорошо описано в книге МакКонела “Совершенный код”. Он приводит там таблицу с разными типами проектов и указывает, какие процессы, приемы и компоненты процессов разработки применимы в разных типах проекта.

Kind of Software	Typical Good Practices		
	Business Systems	**Mission-Critical Systems**	**Embedded Life-Critical Systems**
Typical applications	Internet site Intranet site Inventory management Games Management information systems Payroll system	Embedded software Games Internet site Packaged software Software tools Web services	Avionics software Embedded software Medical devices Operating systems Packaged software
Lifecycle models	Agile development (extreme programming, scrum, timebox development, and so on) Evolutionary prototyping	Staged delivery Evolutionary delivery Spiral development	Staged delivery Spiral development Evolutionary delivery
Planning and management	Incremental project planning As-needed test and QA planning Informal change control	Basic up-front planning Basic test planning As-needed QA planning Formal change control	Extensive up-front planning Extensive test planning Extensive QA planning Rigorous change control
Requirements	Informal requirements specification	Semi-formal requirements specification As-needed requirements reviews	Formal requirements specification Formal requirements inspections
Design	Design and coding are combined	Architectural design Informal detailed design As-needed design reviews	Architectural design Formal architecture inspections Formal detailed design Formal detailed design inspections
Construction	Pair programming or individual coding Informal check-in procedure or no check-in procedure	Pair programming or individual coding Informal check-in procedure As-needed code reviews	Pair programming or individual coding Formal check-in procedure Formal code inspections

Kind of Software	Typical Good Practices		
	Business Systems	**Mission-Critical Systems**	**Embedded Life-Critical Systems**
Testing and QA	Developers test their own code Test-first development Little or no testing by a separate test group	Developers test their own code Test-first development Separate testing group	Developers test their own code Test-first development Separate testing group Separate QA group
Deployment	Informal deployment procedure	Formal deployment procedure	Formal deployment procedure

XP, если коротко, говорит о приоритете кодирования (конструирования) программы перед собственно проектированием, утверждая, что лучший проект – это код (этот тезис излагается в статье “определение программного проекта” Джека Ривза, опубликованной в 1992 году и предвосхитившей появление быстрых методов разработки). Также XP

относительно легко относится к анализу требований, который проводится скорее неформально, поэтапно, эволюционно и присутствует постоянная готовность к изменениям. Здесь, однако, следует отметить, что XP не отвергает проектирование и анализ требований вообще, так как это невозможно.

Другие методики по-иному относятся к доле проектирования и анализа требований. Мне кажется, что прежде всего эта пропорция должна определяться, как указано в таблице, типом проекта.

Таким образом, условно можно рекомендовать быструю 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 возможно и применение традиционного водопада.

Темы 3 – 4
Анализ требований (и планирование)

План
1. С чего начинается – с потребности …
2. Решение о продолжении работы (анализ, исследование).
3. Отличие процессов в плане анализа требований.
4. Список артефактов UP на этапе анализа требований и начала разработки.
5. Прецедентный анализ. Понятие прецедентов и актеров (заинтересованных лиц).
6. Варианты описания прецедентов. Прецеденты и пользовательские истории.
7. Диаграммы прецедентов. Обозначения, назначение и пр.
8. Опасности при анализе прецедентов.
9. Нефункциональные требования. Доп. Спецификация.
10. Видение и словарь терминов.
11. Анализ рисков
12. Планирование в UP и XP.

Как уже было сказано, в разных процессах начальные этапы проекта (определение целей проекта, анализ требований, проектирование и др.), короче говоря, то, что у Мак Конелла называется Prerequisites, то есть – предпосылки, предварительные условия, или можно назвать так “предвариловка” :), занимает разное пропорциональное количество времени. Тем не менее, даже если речь идет об XP, без предварительных этапов не обойтись. МакКонелл выделяет такие предварительные виды деятельности : определение целей проекта, анализ требований, анализ архитектурных решений. У К. Лармана приводится подробное изложение этих этапов с точки зрения UP, поэтому возьмем этот источник в основном за основу...

1. Начало

Все процессы предполагают, что с чего-то проект начинается :). Начинается он с того, что у заказчика появляется какая-то потребность, которая выражается в более или менее четко сформулированных требованиях. Получив начальное описание задачи (в неформализованном виде), разработчик (или коллектив разработчиков) оценивают задачу в целом, объем работ, бюджет и принимают принципиальное решение о начале проекта.

2. Решение о продолжении (начале) проекта

Как принимается такое решение ? Прежде всего, оценивается возможность решения задачи с помощью существующих и доступных для заказчика разработок. Не следует заниматься изобретением велосипеда. Если все-таки существующие средства не подходят, не отвечают требованиям, разработчик определяет, может ли он (они) сделать то, что требуется в принципе, может ли он сделать это лучше, чем существующие аналоги, насколько реальным ему представляются сроки и бюджет. На данном этапе далеко не на все эти вопросы можно ответить, поэтому задача данного этапа – отсечь явно неприемлимые для разработчика проекты. А в принципе не все еще пока ясно даже в принципе. Как писал один автор (Лес Пинтер в книге про FoxPro 2.0) “Когда меня спрашивают, сколько будет стоить программа обработки счетов, я отвечаю – от 100 долл. до миллиона, смотря что Вы хотите.”

В общем случае решение о собственно выполнениии программной разработки может приниматься после достаточно длительных исследований, анализа требований, аналогов. Кроме того, если речь идет не о программном проекте, а о проекте автоматизации (в более общей постановке), то может выполняться анализ требований с последующим созданием технической документации, описывающей использование существующих средств для решения поставленной задачи.

3. Отличие процессов в плане анализа требований

В разных процессах этап анализа требований занимает разное количество времени :

1) В традиционном водопаде он может занимать в среднем до 20-30% времени проекта и заканчивается составлением ТЗ.
2) В унифицированном процессе основные требования анализируются в основном в фазах начала и развития проекта, то есть могут занимать в разных случаях от нескольких недель до нескольких месяцев. Важно отметить, что в UP требования анализируются в несколько приемов , вначале выясняется до 30% процентов требований (причем только 10% описываются подробно), затем еще до 70% - в процессе разработки на этапе развития, остальные – в фазе конструирования системы.
3) В XP начальный этап анализа требований занимает минимальное время, необходимое для начала работ (для составления списка задач, планирования версий и первых 1-2 итераций и пр.).

4. Список артефактов UP на этапе анализа требований и начала разработки.

Заметим, что артефакт — это не документ или диаграмма, а некий результат процесса осмысления, анализа и разработки (с последующей записью результатов).
На этапе начала разработки в UP создаются следующие документы :

- В***и***дение системы – это документ, в котором формулируется общее концептуальное представление о системе, цели ее создания и пр.;
- Прецеденты – описания функциональных требований к системе, о них подробнее ниже;
- Дополнительная спецификация – фактически это список нефункциональных требований;
- Словарь терминов;
- Перечень рисков;
- Перечень прототипов;
- План итерации(й) фазы развития и др.

В XP составляется меньше документов, но также используется список пользовательских историй, список ограничений и прочих требований, и, конечно, составляются планы версий и итераций.

5. Прецедентный анализ. Понятие прецедентов и актеров (заинтересованных лиц).

Прецеденты по существу (неформально) — это рассказы об использовании системы в процессе решения поставленных задач. Вот пример сжатого формата описания прецедента для проекта торгового терминала (POS – Point Of Sale) NextGen из книги К. Лармана:

> **Обработка продажи (process sale).** Покупатель подходит к кассе с выбранными товарами. Кассир с помощью POS-системы регистрирует каждый товар. Система отображает информацию о каждом наименовании товара и вычисляет общую сумму. Покупатель вводит требуемую информацию; система ее верифицирует и регистрирует. Система выполняет инвентаризацию. Покупатель получает товарный чек и покидает магазин с покупками.

Зачастую прецеденты нужно продумывать гораздо детальнее. Но основная идея состоит в исследовании и формулировке функциональных требований путем написания историй "из жизни системы". Эти истории помогают сформулировать различные задачи и представляют собой сценарии использования системы.

Сначала введем некоторые неформальные определения.

Исполнителем или *действующим лицом* (actor) будем называть сущность, обладающую поведением, например, человека (идентифицируемого по роли), компьютерную систему или Организацию, например кассира, систему авторизации кредитных карт или Налоговую службу.

Существует три типа внешних по отношению к разрабатываемой системе исполнителей.

- Основной исполнитель (primary actor) — его задачи выполняются с использованием системы. Примером основного исполнителя является кассир.

  Зачем его идентифицировать? Чтобы определить цели пользователя, на основе которых формулируются прецеденты.
- Вспомогательный исполнитель (supporting actor) — обслуживает систему (например, предоставляет информацию). Примером вспомогательного исполнителя является служба авторизации платежей.

  Зачем его идентифицировать? Чтобы определить внешние интерфейсы и протоколы.
- Закулисный исполнитель (offstage actor) — заинтересован в реализации прецедента, но не является основным или вспомогательным исполнителем. Примером закулисного исполнителя является налоговая служба.

  Зачем его идентифицировать? Чтобы удостовериться, что *все* интересы определены и удовлетворены. Интересы закулисных исполнителей обычно не очевидны и их легко упустить из виду, если не идентифицировать их в явной форме.

Сценарий (scenario) — это специальная последовательность действий или взаимодействий между исполнителями и системой. Его иногда также называют *экземпляром прецедента* (use case instance). Это один конкретный сценарий использования системы либо один проход прецедента, например, сценарий успешной покупки товаров за наличный расчет, либо сценарий неудачного завершения покупки из-за прерванной транзакции по обработке данных кредитной карточки.

Неформально, *прецедент* (use case) — это набор взаимосвязанных успешных и неудачных сценариев, описывающий использование системы исполнителем для решения одной из задач. Например, рассмотрим свободный формат прецедента, включающего некоторые альтернативные сценарии.

Возврат товара (Handle Returns)

Основной успешный сценарий. Покупатель подходит к кассе с товарами, подлежащими возврату. Кассир использует POS-систему для регистрации каждого возвращаемого товара...
Альтернативные сценарии.
Если идентификатор товара в системе не обнаружен, система уведомляет об этом кассира и предлагает ему вручную ввести идентификационный код (возможно, штрих-код поврежден и его сложно считать).
Если у системы возникают сложности при коммуникации с внешней системой вычисления налога,...

Несколько другое, но подобное определение прецедента приводится в RUP. *Прецедент* — это набор сценариев использования, в котором каждый экземпляр сценария представляет собой последовательность действий, выполняемых системой для достижения *ощутимого для конкретного исполнителя результата*.

Фраза *"ощутимый результат"* несколько туманна, но очень важна, поскольку она указывает на то, что поведение системы должно быть ощутимо для пользователя.

Основное внимание при описании прецедента нужно сконцентрировать на вопросе: "Как использование системы обеспечивает ощутимый для пользователя результат или решает его задачу?", а не на обдумывании системных требований в терминах свойств или функций.

На первый взгляд, требование обеспечения ощутимого результата может показаться очевидным. Однако в сфере программного обеспечения известно множество неудачных проектов, которые провалились из-за невыполнения реальных задач пользователей. К такому отрицательному результату может привести подход к описанию системных требований в виде списка свойств и функций системы, поскольку он не заставляет заинтересованных лиц рассматривать требования в более широком контексте достижения некоторого ощутимого

результата или некоторой цели. В отличие от этого подхода, в контексте прецедентов свойства и функции системы ориентированы на достижение цели.

В этом состоит основная идея, сформулированная Якобсоном относительно прецедентов: требования должны быть направлены на получение ощутимого результата и достижения целей.

Итак, прецеденты — это требования (хотя и не все требования). Некоторые считают требованиями только список функций и свойств типа "система должна...". На самом деле это не так. Ключевая идея использования прецедентов как раз и состоит (обычно) в снижении роли списка требований в старом понимании этого слова. Теперь в качестве функциональных требований выступают сами прецеденты.

6. Варианты описания прецедентов. Прецеденты и пользовательские истории.

Прецеденты типа "черный ящик" и системные обязанности

Прецеденты типа "черный ящик" (black-box use cases) — это самый типичный и рекомендуемый тип прецедентов. Они не описывают внутреннюю работу системы, ее компоненты или дизайн. Наоборот, системе вменяются некоторые *обязанности* (responsibilities). Этот метафорический термин широко применяется в объектно-ориентированном проектировании: программные элементы имеют обязанности и взаимодействуют с другими элементами со своими обязанностями.

Определяя обязанности системы через прецеденты типа "черный ящик", можно указать, *что* должна делать система (функциональные требования), не расписывая, *как* это делать (не выполняя проектирование). Вообще, термины "анализ" и "проектирование" зачастую сводятся к вопросам "что" и "как". Это важные вопросы в хорошей программной разработке. В процессе анализа требований нужно избегать принятия решений "как", а описывать лишь внешнее поведение системы как черного ящика. Позднее, на этапе проектирования, создается решение, удовлетворяющее разработанной спецификации. (Пример с сохранением данных в БД с помощью SQL).

Степень формализации

Прецеденты описываются в различных форматах, в зависимости от потребностей. Помимо типов "черного ящика" и "белого ящика", выделяют несколько степеней формализации описания прецедентов.

- *Сжатый* — аннотация в виде одного абзаца. Обычно она описывает только главный успешный сценарий. Пример такого описания приведен выше для прецедента Оформление продажи (Process Sale).
- *Свободный* — неформальный стиль описания. Описание прецедента занимает несколько абзацев и охватывает различные сценарии. Примером такого описания является рассмотренный выше прецедент Возврат товара.
- *Развернутый* — наиболее подробный стиль описания. При таком подходе детально описываются все шаги и варианты развития сценария, а также предусловия и результаты.

Рассмотрим пример развернутого описания прецедента для системы NextGen.

Пример развернутого описания прецедента **Оформление продажи**

Развернутые описания прецедентов структурированы и содержат большое количество деталей. Их полезно использовать для углубления понимания целей, задач и требований. Для примера POS-системы NextGen такие описания можно обсуждать на семинарах по определению требований на начальной стадии проекта вместе с системным аналитиком, экспертами предметной области и разработчиками.

Формат usecases.org

Для развернутого описания прецедентов существуют различные шаблоны

форматирования. Однако чаще всего используется шаблон, приведенный на Web-узле www.usecases .org. Этот стиль проиллюстрирован в следующем примере.

Этот пример детального описания прецедента относится к рассматриваемой в книге К.Ламана системе NextGen и отражает множество типичных элементов и вопросов.

Прецедент П1. Оформление продажи

Основной исполнитель. Кассир.

Заинтересованные лица и их требования

- Кассир. Хочет точно и быстро ввести данные, не допуская ошибок в платеже, поскольку недостача вычитается из его зарплаты.
- Продавец. Хочет получить свои комиссионные от продажи.
- Покупатель. Хочет купить товары и быстро оформить покупку с минимальными усилиями. Хочет получить подтверждение факта покупки для возможного возврата товара.

10) Компания. Хочет аккуратно записать транзакцию и удовлетворить интересы покупателя. Хочет удостовериться, что служба авторизации платежей зафиксировала все данные о платеже. Заинтересована в обеспечении устойчивости к сбоям; хочет продолжать регистрировать продажи, даже если серверные компоненты (например, служба удаленной проверки кредитоспособности) недоступны. Хочет автоматически обновлять бухгалтерскую документацию и вести складской учет.
11) Государственные налоговые службы. Хотят получать налог от каждой продажи. Таких служб может быть несколько, в том числе национальная и местная.

Предусловия. Кассир идентифицирован и аутентифицирован.

Результаты (постусловия). Данные о продаже сохранены. Налоги корректно вычислены. Бухгалтерские и складские данные обновлены. Комиссионные начислены. Чек сгенерирован. Авторизация платежа выполнена.

Основной успешный сценарий (или основной процесс)

1) Покупатель подходит к кассовому аппарату POS-системы с выбранными товарами.
2) Кассир открывает новую продажу.
3) Кассир вводит идентификатор товара.
4) Система записывает наименование товара и выдает его описание, цену и общую стоимостью Цена вычисляется на основе набора правил.

Кассир повторяет действия, описанные в пп. 3-4, для каждого наименования товара.

5) Система вычисляет общую стоимость покупки с налогом.
6) Кассир сообщает покупателю общую стоимость и предлагает оплатить покупку.
7) Покупатель оплачивает покупку, система обрабатывает платеж.
8) Система регистрирует продажу и отправляет информацию о ней внешней бухгалтерской системе (для обновления бухгалтерских документов и начисления комиссионных) и системе складского учета (для обновления данных).
9) Система выдает товарный чек.
10) Покупатель покидает магазин с чеком и товарами (если он что-то купил).

Расширения (или альтернативные потоки)

*а. При каждом выходе системы из строя.
Для ввода системы в строй и корректной обработки платежа нужно обеспечить восстановление всех транзакций и событий с любого шага сценария.

1. Касссир перезапускает систему, регистрируется и предлагает восстановить предыдущее состояние.
2. Система восстанавливает предыдущее состояние.

2а. Система определяет аномалию, повлекшую сбой.

1. Система уведомляет об ошибке кассира, регистрирует ошибку и переходит в начальное состояние.
2. Кассир начинает новую продажу.

3а. Неправильный идентификатор.

1. Система уведомляет об ошибке и отменяет ввод данного наименования товара.

3б. В рамках одной категории существует несколько разных наименований товара и идентифицировать конкретное наименование не нужно (например, 5 пакетов леденцов).

1. Кассир может ввести идентификатор категории товара и количество единиц.

3-6а. Покупатель просит кассира отменить покупку одного из товаров.

1. Кассир вводит идентификатор товара для удаления из продажи.
2. Система отображает обновленную промежуточную стоимость.

3-6б. Покупатель просит кассира отменить продажу.

1. Кассир отменяет продажу.

3-6в. Кассир приостанавливает продажу.

- Система записывает сведения о продаже таким образом, чтобы они были доступны с любого терминала POS-системы.

4а. Сгенерированная системой цена товара не устраивает покупателя (например, у него есть дисконтная карта и он рассчитывает на более низкую цену товара).

- Кассир вводит команду об изменении цены.
- Система вычисляет новую цену.

5а. Система выявляет сбой при коммуникации с внешней службой вычисления налога.

Система перезапускает службу с данного узла POS – системы и продолжает работу.

1а. Система определяет, что служба не перезапускается.

Система сигнализирует об ошибке.

Кассир сожет вручную вычислить и ввести сумму налога либо отменить продажу.

5б. Покупатель сообщает о положенной ему скидке (например, для сотрудников данного предприятия или постоянных покупателей).
1. Кассир отправляет запрос на скидку.
2. Кассир вводит идентификационные данные покупателя.
3. Система предоставляет сумму скидки, вычисленную на основе дисконтных правил.

…

Специальные требования

12) Сенсорный экран с интерфейсом пользователя для большого плоского монитора. Текст должен быть виден с расстояния один метр.
13) Отклик службы авторизации в 90% случаев приходит в течении 30 секунд.
14) Каким-то образом нужно обеспечить робастное восстановление информации в случае сбоя при доступе к удаленным службам, таким как система складского учета.
- Возможность локализации (представления на различных языках) отображаемого текста.
- Возможность добавления новых бизнес-правил на шагах 3 и 7 в процессе функционирования системы.

Список технологий и типов данных

3а. Идентификатор товара считывается со штрих-кода (при наличии последнего) лазерным сканером или вводится с клавиатуры. 36. Идентификатор товара может определяться по схемам кодирования UPC, EAN, JAN или SKU. 7а. Информация об открытом кредите вводится с помощью считывающего устройства или с клавиатуры. 76. Подпись при оплате чеком ставится на бумажном документе. Однако ожидается, что в течение двух лет большинство покупателей будут требовать цифровые устройства считывания подписи.

Частота использования: почти постоянно.

Открытые вопросы

- Изучить законодательство по налогообложению.
- Исследовать вопрос восстановления удаленных служб.
- Какая настройка потребуется для различных типов магазинов.
- Должен ли кассир снимать кассу при выходе из системы.
- Может ли пользователь сам использовать устройство считывания данных с карточки или это должен делать кассир.

Этот прецедент скорее является иллюстративным, чем исчерпывающим (хотя и основывается на реальных требованиях к POS-системе). Тем не менее, он описан достаточно подробно и позволяет составить реалистичное представление о развернутом формате описания прецедентов и их сложности. Этот пример может служить моделью для многих других прецедентов.

Представление в виде двух колонок

Некоторые предпочитают оформлять прецеденты в виде двух колонок, обращая внимание на факт взаимодействия исполнителей и системы.

Прецедент П1. Оформление продажи

Основной исполнитель:...
... как и ранее...

Основной успешный сценарий

Действие исполнителя
1. Покупатель подходит к кассовому аппарату POS-системы с выбранными товарами.
2. Кассир открывает новую продажу.

3. Кассир вводит идентификатор товара.

Кассир повторяет действия, описанные в пп. 3-4 для каждого наименования товара.

Отклик системы (2 колонка)

4. Система записывает наименование товара и выдает его описание, цену и общую стоимость. Цена вычисляется на основе набора правил.
5. Система вычисляет общую стоимость покупки с налогом.

6. Кассир сообщает покупателю общую стоимость и предлагает оплатить покупку.

Задачи и рамки прецедента

Как выделить прецедент? Зачастую определить правильный (а точнее, полезный) прецедент очень сложно. Каждую задачу можно рассматривать на разных уровнях детализации, начиная от конкретных простых действий и заканчивая деятельностью на уровне предприятия.

Прецедент — это не один маленький шаг, такой, например, как удаление товара или печать документа. Основной сценарий прецедента обычно включает пять-десять шагов. Описываемый сценарий выполняется не в течение нескольких дней или сеансов, как, например, переговоры с поставщиками. Это задача, выполняемая в течение одного сеанса. Реализация прецедента может длиться от нескольких минут до нескольких дней. Как и при определении унифицированного процесса, основное внимание уделяется получению ощутимого (измеримого) результата и переходу системы и данных в устойчивое состояние.

Типичной ошибкой является рассмотрение прецедентов на слишком низком уровне, когда сценарий выполняется за один шаг и по существу является некоторой подфункцией или подзадачей в рамках элементарного бизнес-процесса.

Роль прецедентов в проекте

Прецеденты играют жизненно важную роль при реализации унифицированного процесса, поскольку вся разработка в рамках этого подхода осуществляется *под управлением прецедентов* (use-case driven development). Это означает следующее.

- Требования в основном формулируются при описании прецедентов (в модели прецедентов). Остальные требования (если таковые существуют) являются либо техническими (например, список функций), либо второстепенными.
- Прецеденты — важный этап итеративного планирования. На каждой итерации реализуются некоторые сценарии или целые прецеденты. Поэтому описания прецедентов вносят существенный вклад в оценивание результата.
- Разработка приложения состоит в реализации прецедентов. То есть группа разработчиков продумывает способы взаимодействия объектов или архитектуру подсистем для реализации прецедентов.
- Прецеденты служат основой приемки программы и приемочного тестирования.

Когда составляются прецеденты и насколько детально.

Вначале формулируются прецеденты в сжатом виде (5 мин. на один, но большинство, потом – 30% подробно, затем, к концу фазы развития – до 70%)

Группа технической разработки приступает к созданию ядра системы в тот момент, когда детализированы лишь 10% требований, далее следует намеренная задержка, и формулировка требований откладывается до конца первой итерации стадии развития.

В этом состоит ключевое отличие от однопроходного процесса — промышленная разработка ядра системы начинается достаточно быстро, задолго до полного завершения формулировки требований.

Обратите внимание, что в конце первой итерации стадии развития проводится второй семинар по формулировке требований, по окончании которого 30% прецедентов описаны подробно. При таком планировании можно учесть обратную связь и оценки реализованной части системы. Обратная связь поступает от пользователей, специалистов по тестированию и позволяет "познать непознанное". То есть в процессе разработки системы очень быстро возникают вопросы, требующие немедленного решения.

7. Диаграммы прецедентов. Обозначения, назначение и пр.

В языке UML существует система, обозначений для диаграммы прецедентов, иллюстрирующей имена прецедентов, исполнителей и взаимосвязи между ними. (Пример).

Диаграмма прецедентов — это отличное изображение системного контекста, поскольку она отображает границы системы, внешние для системы понятия и способы использования системы. Она подытоживает поведение системы и ее исполнителей.

Диаграммы прецедентов и их взаимосвязей имеют второстепенное значение при работе над прецедентами. Прецеденты — это текстовые документы. Работать над прецедентами — значит составлять текстовые описания.

8. Опасности при составлении прецедентов.

О прецедентах написано достаточно много. Но все равно существует опасность, что умные творческие люди могут слишком усложнить изначально простую идею. Зачастую начинающие специалисты по описанию прецедентов слишком увлекаются созданием диаграмм прецедентов, описанием их взаимосвязей, составлением пакетов прецедентов, рассмотрением необязательных атрибутов и т.д., а не написанием историй. Сила механизма прецедентов состоит в возможности масштабировать уровень сложности и формальности описания в зависимости от реальных потребностей.

Диаграммы последовательности.

9. Нефункциональные требования (дополнительная спецификация)

Перечислим вопросы, которые могут включаться в доп. Спецификацию :

Функциональность
{Имеющая отношение ко многим прецедентам)

Регистрация событий и обработка ошибок

Подключаемые бизнес-правила

Безопасность

Удобство использования

Человеческие факторы

Надежность

Возможность восстановления информации

Производительность

Возможности поддержки

Адаптация системы

Конфигурирование

Ограничения

Приобретаемые компоненты

Бесплатные компоненты на основе открытого кода

Интерфейсы

Важные интерфейсы и аппаратные средства

Программные интерфейсы

Бизнес-правила

Имя	Правило	Возможность изменения	Источник

Вопросы законодательства

Информация из предметной области

Ценовая политика

Обработка платежей по кредитной и дебитной карточке

Вычисление налогов

Идентификаторы товаров (UPC, EAN, SKU, штрих-коды и сканеры)

В дополнительной спецификации содержатся требования, ограничения и другая информация, не вошедшая в описание прецедентов или словарь терминов, включая атрибуты качества и специальные требования. Заметим, что требования, связанные с прецедентами, должны быть представлены в описаниях прецедентов в разделе "Специальные требования", однако некоторые предпочитают также включать их в дополнительную спецификацию. В дополнительную спецификацию можно включать следующие элементы.

- Требования согласно модели FURPS+ — функциональные, требования к удобству использования, надежности, производительности и возможности поддержки.
- Отчеты.
- Ограничения на аппаратные и программные средства (операционные и сетевые системы и т.д.).
- Ограничения, накладываемые на процесс разработки (например, процесс или средства разработки).
- Другие ограничения проектирования или реализации.
- Международные соглашения (единицы измерения, языки и т.д.).
- Документация (пользовательская, руководство по установке и администрированию) и справочная информация.
- Соглашения о лицензировании или другие юридические соглашения.
- Разбиение на пакеты.
- Стандарты (технические, обеспечения качества и безопасности).
- Физические требования к окружению (например, температурный режим эксплуатации или ограничения на вибрацию).
- Операционные требования (например, способ обработки ошибок, частота архивации).
- Информация из предметной области (например, о полном цикле обработки платежа по кредитной карточке).

10. В*и*дение

Введение

Позиционирование

Экономические предпосылки

Формулировка проблемы

Место системы

Заинтересованные лица

Необходимо определить, для кого предназначена система и каковы проблемы заинтересованных лиц.

Демографические особенности рынка...

Заинтересованные лица, не являющиеся пользователями системы...

Пользователи системы...

Основные задачи высокого уровня и проблемы заинтересованных лиц

Необходимо объединить информацию из списка исполнителей и задач, а также из раздела описания прецедентов, отражающего потребности заинтересованных лиц.

Задачи уровня пользователя
Сюда можно включить список исполнителей и их задач, разработанный в процессе моделирования прецедентов, либо более сжатую информацию. Пользователи (и внешние системы) используют данную систему в таких целях.

- *Кассир.* Оформляет продажи, возврат товаров, регистрирует выручку.
- *Системный администратор.* Управляет пользователями, безопасностью и системными таблицами.
- *Менеджер,* Осуществляет запуск и завершает работу системы.
- *Система анализа торговой деятельности.* Анализирует данные о продажах.
- ...

Окружение...

Обзор

Перспективы продукта

Преимущества системы
Подобно перечню исполнителей и их задач, в этой таблице указаны задачи, их решения и преимущества, однако на более высоком уровне, чем при описании прецедентов.
Здесь описывается основное значение и отличительные свойства продукта.

Предположения и зависимости... Стоимость и

ценообразование... Лицензирование и установка...

Основные свойства системы

Как было упомянуто выше, свойства системы описываются сжато путем перечисления основных функций.

Оформление продаж.

Авторизация платежей (по кредитной или дебитной карточке, чеком), Системное администрирование и управление пользователями, безопасностью, таблицами констант и кодов и т.д.

Автоматический переход в автономный режим работы при выходе из строя внешних систем. Транзакции в реальном времени на основе промышленных стандартов с внешними системами, включая бухгалтерскую систему, систему складского учета, учета человеческих ресурсов, вычисления налогов, службы авторизации платежей.

Определение и выполнение настраиваемых бизнес-правил в фиксированных точках выполнения сценариев.

Другие требования и ограничения

Ограничения для процесса проектирования, удобства использования, надежности, производительности, перечень документации и т.д. описаны в дополнительной спецификации и модели прецедентов.

О порядке разработки артефактов говорить неуместно. Различные артефакты, относящиеся к требованиям, создаются параллельно. Тем не менее, можно порекомендовать такую последовательность разработки.

1. Создайте краткий черновой вариант документа "Видение".
2. Идентифицируйте задачи пользователей и соответствующие прецеденты.
3. Опишите некоторые прецеденты и приступите к разработке дополнительной спецификации.

4. На основе полученной информации уточните документ "Видение".

11) Развитие проекта. Риски.

Фаза развития — это первая последовательность итераций, в течение которых команда разработчиков выполняет серьезные исследования базовых элементов архитектуры, **их** реализацию (написание программного кода и его тестирование), определяет для себя большинство требований и снимает вопросы, связанные с наиболее высокими рисками. В контексте UP термин "риск" имеет экономическое значение. То есть на ранних этапах реализации проекта должны разрабатываться сценарии, достаточно важные, но не обязательно сопряженные с техническим риском.

Фаза развития зачастую состоит из 2-4 итераций, каждая из которых длится от двух до шести недель. Время каждой итерации жестко фиксировано. Если поставленные задачи сложно выполнить к назначенному сроку, то некоторые требования переносятся на последующие итерации, чтобы данная итерация завершилась вовремя и в результате был получен устойчивый и протестированный код.

Фаза развития — это не стадия проектирования или подготовки к реализации, как было принято в рамках каскадного процесса (в стиле "водопада").

На этой стадии создаются не прототипы, а полностью разрабатывается некоторый фрагмент системы. В некоторых описаниях UP для представления части системы используется термин *архитектурный прототип* (architectural prototype), который может быть неверно истолкован. В данном случае прототип — это не экспериментальный образец, а рабочее подмножество окончательной системы. Эту часть системы также называют *исполняемой архитектурой* (executable architecture) или *архитектурной основой* (architectural baseline).

Коротко о фазе развития

Построение базовой архитектуры, разрешение высоких рисков, определение большинства требований и оценка общего графика реализации и необходимых ресурсов.

Требования и итерации систематизируются в соответствии с рисками, границами и критичностью.

- *Риск* (risk) — это техническая сложность или другой фактор, например, отсутствие информации о необходимых затратах или ресурсах.
- *Границы* (coverage) — на начальных итерациях нужно определить все основные части системы, т.е. выполнить реализацию множества компонентов "не вглубь, а вширь".
- *Критичность* (criticality) — требуется реализовать функции, имеющие важное значение для системы.

Список артефактов фазы развития **:**

Артефакт	Комментарий
Модель предметной области	Она представляет собой визуализацию понятий предметной области, напоминающую статическую модель сущностей предметной области
Модель проектирования	Это набор диаграмм, описывающих логику проектного решения. Сюда относятся диаграммы программных классов, диаграммы взаимодействия объектов, диаграммы пакетов и т.д.
Описание программной архитектуры	Это документ, в котором рассмотрены основные архитектурные моменты и способы их реализации. В нем приводятся основные идеи проектного решения и обосновывается их целесообразность для данной системы
Модель данных	Включает схему базы данных и стратегию отображения объектов в необъектное представление
Модель тестирования	Описание задач и способов тестирования
Модель реализации	Это реальная реализация — с исходным кодом, исполняемыми файлами, базой данных и т.д.
Прототипы интерфейса пользователя	Описание интерфейса пользователя, способов навигации и т.д.

Тема 4

Экстремальное планирование

«Готовясь к драке, я всегда обнаруживал, что планы совершенно бесполезны. А вот без планирования не обойтись».
Д. Эйзенхауэр

Экстремальное программирование стоит, если так можно выразиться, на 3 китах : планирование, тестирование, рефакторинг. (Другие практики тоже, однако, забывать не стоит !). Как же выглядит экстремальное планирование ? Подробно о нем рассказано в книге К.Бека, которая так и называется : экстремальное планирование. Естественно, что мы не можем за короткое время лекции рассмотреть ее содержание, но можно составить представление о самом процессе.

Для чего выполняют планирование в XP :
- 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.

Еще короче :
- чтобы заниматься самыми важными делами на данный момент;
- чтобы согласовывать свои действия с действиями других людей (сдругими разработчиками при командной разработке, с заказчиком – всегда);
- чтобы знать, что делать при возникновении нештатных ситуаций при выполнении двух предыдущих пунктов.

Итак, мы провели анализ (начальный этап), возможно, подумали об архитектуре нашего приложения (собственно, только эскизно). Да, еще, возможно, мы написали несколько прототипов (программок, которые можно набросать 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 !

Вообще у экстремального планирования можно выделить ряд ключевых особенностей :

15)Итеративность разработки.
16)Планирование выполняется постоянно. План постоянно подвержен изменениям. «Ловушка планирования» - ?.
17)Составляются разные по степени подробности и временному интервалу планы : план версий, план итераций, план текущей итерации план масштаба дня и т.д.
18)Планы версий и итераций более всего корректируется на границах итераций и версий, хотя может корректироваться и в процессе итераций. Само планирование выполняется ежедневно.
19)Одним из ключевых моментов планирования является измерение производительности команды разработчиков. Производительность измеряется в задачах, решенных за «идеальный день», соответственно план расписан в идеальных днях.
20)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.**
21)Одной из ключевых идей планирования (и всей книги) является следующая простая мысль: 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. То есть, разработчики не говорят – «нам не хватает времени», они говорят «у нас слишком много дел». Речь не идет об обмане и лентяйстве – речь идет только о трезвой оценке своих возможностей и борьбе со стрессом и переработкой.

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

23) Список задач для выполнения управляется также рисками, то есть борьбой со страхами разработчика (да и заказчика тоже). Приоритеты определяются с учетом рисков и пожеланий разработчиков.

Идеальный день – это день, полностью посвященный (8 часов) решению задач, при котором вы себя хорошо чувствовали, у вас была нормальная работоспособность и вы не отвлекались на другие задачи, не связанные с проектом.

Риски и страхи – выбор задач для итераций. Роль заказчика.

Итеративность – ключ к успеху.

Четыре переменные планирования : скорость, время, качество, объем работ. (управлять только последними двумя).

Планирование от достигнутого.

Пример плана версий, плана итераций, плана на текущий день. (космоагенство)

Контроль за ходом итерации.

Вычеркивание задач (заказчик).

Стоячие собрания.

Метафоры планирования

«Ловушка планирования», «Вождение автомобиля», «Магазин», «Вчерашняя погода».

Темы 5-6. Классификация. Моделирование предметной области (домена).

План:

1. Место и роль моделирования предметной области.
2. Понятие класса, объекта, атрибутов, методов, **инкапсуляция, наследование, полиморфизм** и пр.
3. Отношения между объектами (ассоциации).
4. Диаграммы классов UML.
5. Процесс классификации. Проектная процедура по Страуструпу.
6. Пример классификации из Лармана.
7. **Какие еще диаграммы нужны (кооперации, активности, возможно даже – состояния)**?

1. Место и роль моделирования предметной области.

Модель предметной области (домена) составляется на этапе развития проекта (по UP). В ней приводятся классы понятий реального мира. Модель предметной области – визуальное представление концептуальных классов или объектов реального мира в терминах предметной области.

Пример модели предметной области ()

Данная модель - это визуальный словарь абстракций. Это не модель программных классов. Основное отличие ООА от структурного анализа состоит в декомпозиции проблемы на понятия, а не на функции.

Ракурсы модели (Rational Rose):

3. Концептуальный аспект
4. Аспект спецификации
5. Аспект реализации

2. Основные понятия ООП.

Неформально :

Объект: некая сущность реального мира или концептуальная сущность (характеризуется состоянием, поведением, и индивидуальностью). Состояние определяется **атрибутами** (память состояний объекта). Поведение – **методами и операциями.** Индивидуальность – каждый объект индивидуален.

Класс – шаблон для объектов или тип объектов (упрощенно).

Ассоциация – двунаправленная семантическая связь между классами.

Инкапсуляция – дословно “сокрытие”. Можно толковать инкапсуляцию двояко : это, с одной стороны, абстракция “активных данных”, то есть соединение в одном объекте и данных, и методов их обработки, с другой стороны – сокрытие данных (полей, атрибутов) с помощью методов доступа и других методов, с помощью которых и можно только менять атрибуты и таким образом состояние объекта. Единственный способ изменить состояние объекта – **передача сообщений**.

Наследование – установление отношения “родитель-потомок” между классами. Класс – наследник обладает всеми свойствами наследуемого класса, то есть списком атрибутов и методов, но как правило добавляет что-то свое. Наследник является наследуемым классом тоже, поэтому такой тип оношения называют отношение “is-a”.

Полиморфизм - (“много форм” – дословно) – определяет различные формы реализации одноименного действия. С одной стороны, полиморфизм – это наличие в

иерархии методов с одинаковой **сигнатурой**, которые по-разному выполняются разными наследниками (в т.ч. - простой статический полиморфизм), с другой – вызов метода для неявно определенного объекта (указывается тип базового класса, который во время выполнения может подменяться разными наследниками), то есть данное понятие связано с поздним связыванием.

Делегирование – передача сообщения известному объекту для выполнения части (или всех) собственных обязанностей. (?)

3. Основные типы отношений

Ассоциация – первый тип отношений, связан с передачей сообщений путем **делегирования** (понятие ?).

Агрегация – отношение типа "часть-целое" (логическое включение)

Композиция – отношение типа "часть-целое" (физическое включение)

Отношения агрегации и композиции в основном различаются по критерию времени жизни объектов и возможности их отдельного существования.

Зависимость – самый слабый тип отношений. Обычно связан с косвенной ссылкой (отсутствие ссылки в списке атрибутов класса, например, присутствует в списке параметров методов, создается внутри тела метода, возвращается при каком-то вызове и пр.)

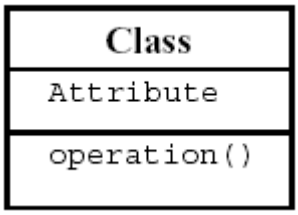
Наследование – второй тип отношений, связан с передачей сообщений путем передачи по иерархии.

Реализация – наследование от абстрактного класса (интерфейса) с реализацией его неопределенных методов (чисто виртуальных функций в C++)**.**

Первый раз упоминание о предпочтении делегирования перед наследованием. Наследование от абстрактных классов. Отличия классов домена от программных.

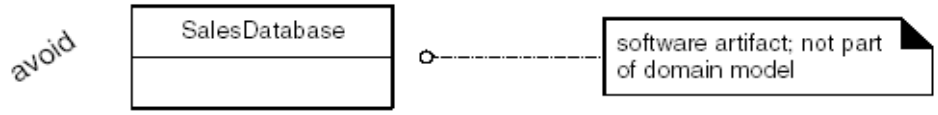
4. Обозначения UML для диаграмм классов

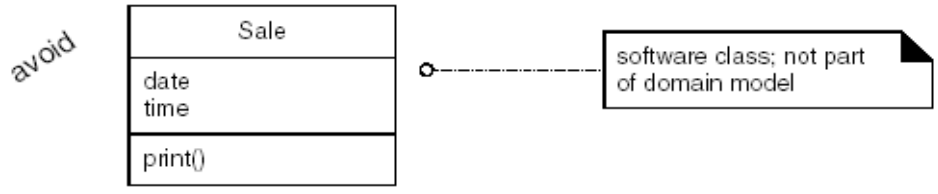
а) объекты и классы



Абстрактный класс : класс с абстрактными методами.

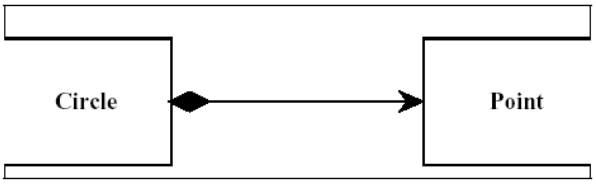
Нюанс : программные классы и классы предметной области



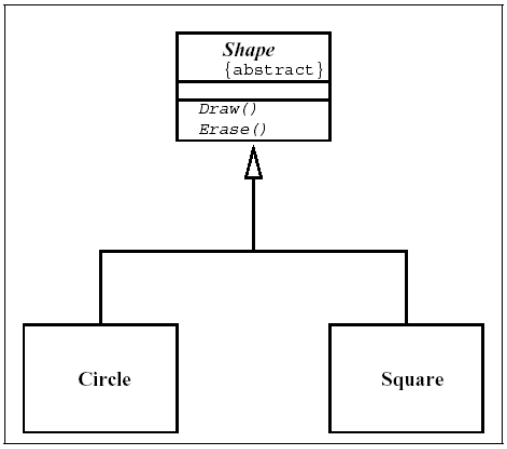


б) Отношения и ассоциации
Типы отношений: а) ассоциация, б) наследование, в) зависимость (более слабое отношение).

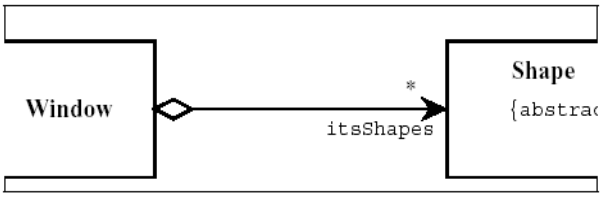
Композиция:



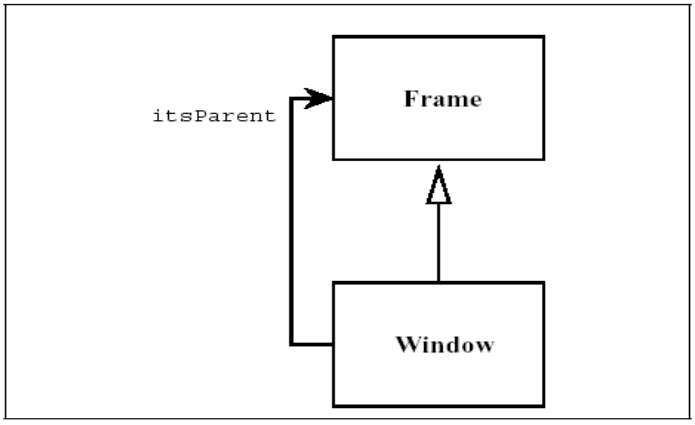
Наследование (отношение обобщения / специализации)



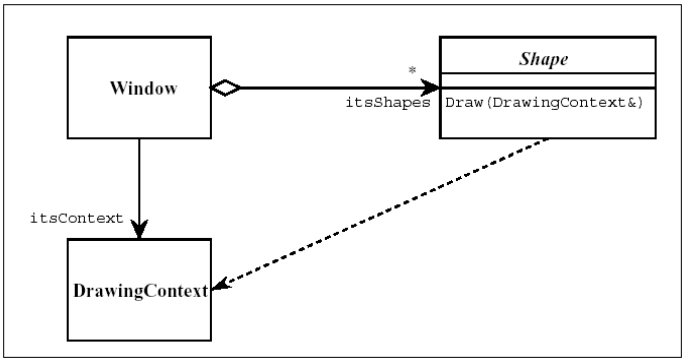
Агрегация



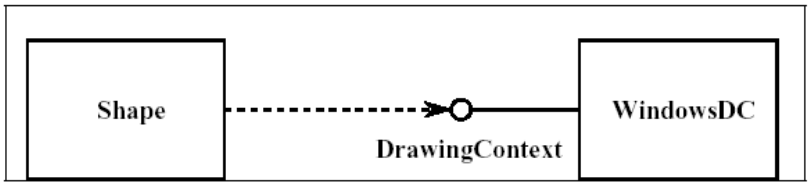
Ассоциация
(родительское окно и дочерние окна …)



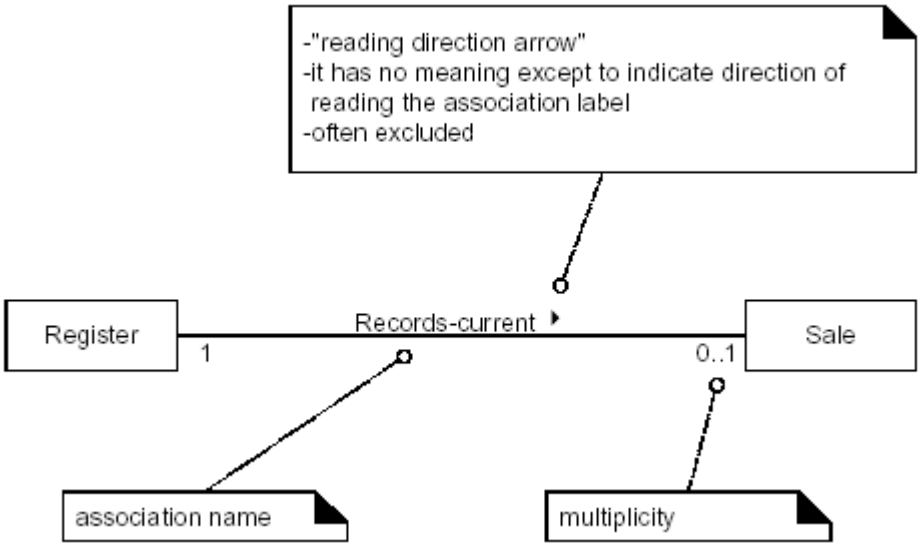
Зависимость



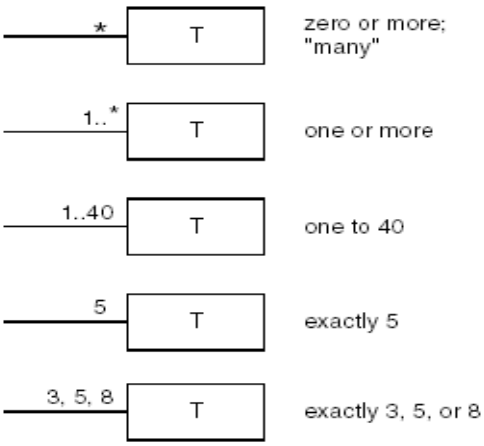
+ реализация



б) как показывается (в т. ч. навигация)



в) арность ассоциаций :



5. Процесс классификации. Проектная процедура по Страуструпу.

Классификация – неформально это процесс выделения классов, определения их атрибутов и отношений между ними. Общий алгоритм классификации примерно такой :

1. Выделить классы.
2. Определить отношения между классами
3. Определить атрибуты.
4. Уточнить определения классов и отношений.

Данная процедура выполняется в контексте создания модели предметной области (домена). При этом не рассматриваются методы классов, так как модель представляет только визуальный словарь абстракции и не обязательно соответствует однозначно модели реализации.

По Б.Страуструпу (по учебнику В.А. Камаева и В.В. Костерина) :

1. Выделение понятий и установление основных связей между ними.
2. Уточнение классов с определением наборов операций для каждого
3. Уточнение классов с точным определением их зависимостей от других классов. Добавление наследования.
4. Задание интерфейсов классов. Более точное определение отношений. Разделение методов на общие и защищенные.

Данная процедура предполагает плавный переход от модели предметной области (концептуальной) к модели реализации.

У Лармана рассматривается несколько иной переход – рассмотрение системных операций в контексте прецедентов, разработка диаграмм последовательности, откуда собственно и формируются методы.

В XP модель предметной области является основой для быстрого эскизного проекта классов с последующей доработкой в процессе TDD и рефакторинга (мое представление !).

Основными подходами из возможных являются:

1. Использование списка категорий концептуальных классов.
2. На основе выделения существительных.

5.1. Список категорий:

Физические или материальные объекты
Спецификации и описания
Места
Транзакции
Элементы транзакций
Роли людей
Контейнеры других объектов
Содержимое контейнеров
Компьютеры или внешние технические системы
Абстрактные понятия
Организации
События
Процессы
Правила и политики
Каталоги
Записи деятельности
Документы

Финансовые инструменты и службы

5.2. Существительные

Правило: если объект в реальном мире не является числом, датой или текстом, то есть атомарным типом, то это, скорее всего, класс. Иначе существительное или словосочетание

5.3. Поиск ассоциаций

Стандартные категории ассоциаций :

A является частью B (логической /физической)
A содержится в B.
A является описанием B
A является элементом транзакции B.
A известен в B
A является членом B
A является подразделением B
A использует / управляет B
A взаимодействует с B
A следует за B
A является собственностью B
A связано с B

Правила:

1) Лучше меньше, чем больше.
2) Выделять наиболее важные ассоциации (длительные).
3) Классы важнее ассоциаций.

6. Пример(ы) моделей предметной области. Из Лармана…

Это пример модели предметной области (части модели) POS-системы. Здесь использованы следующие концептуальные классы : **Item** – товар, **SalesLineItem** – строка товара в списке покупок, **Sale** – Продажа (список покупок), **Payment** – Платеж, **Store** – **Магазин (склад)**, **Register** – касса (кассовый аппарат), **Product Description** - Описание товара, **Product Catalog** – Список товаров, Диаграмма отражает следующие ассоциации: Товар хранится на Складе (в Магазине), в Магазине имеется несколько Кассовых апапаратов, в Кассовом аппарате регистрируется Покупка, которая состоит из нескольких Строк товара, которые отражают сведения о покупке определенного количества Товаров. Покупка оплачивается Платежом. Эту информацию можно почерпнуть из описания прецедентов, например :

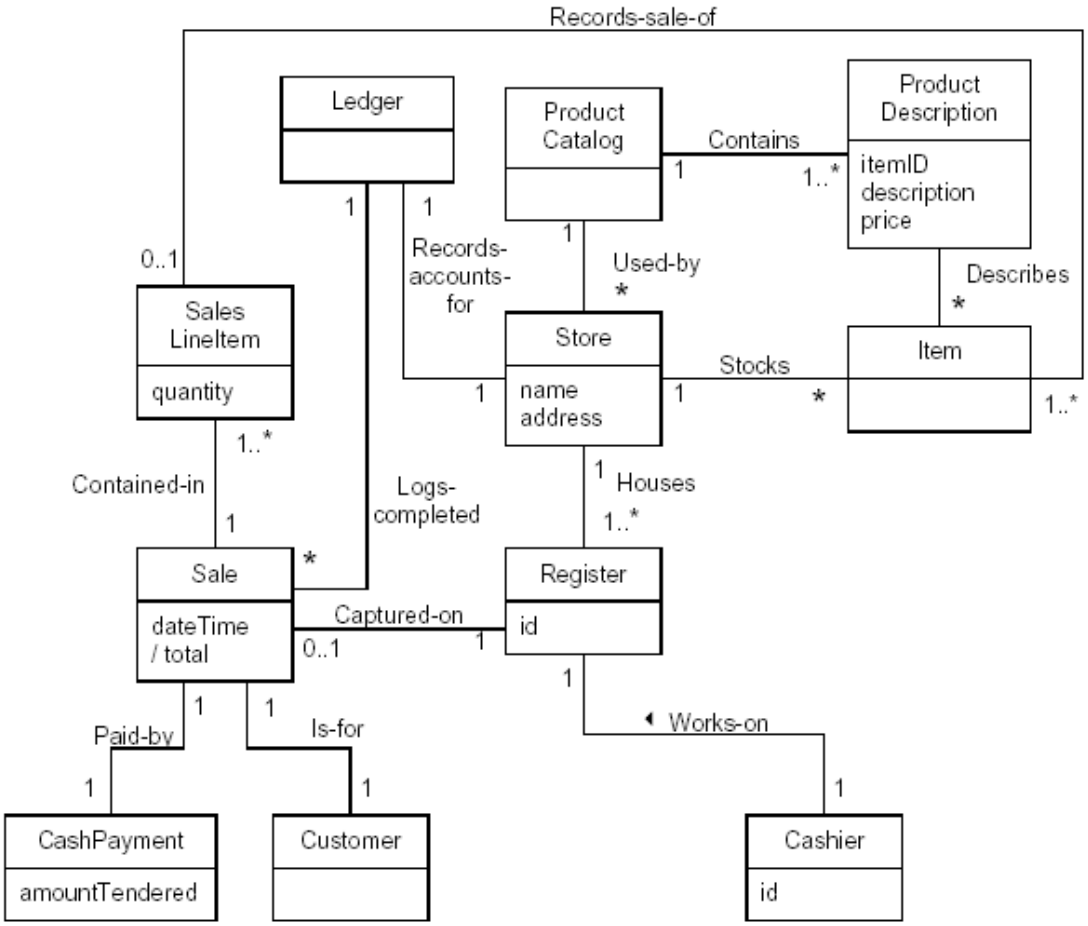
Основной успешный сценарий (или основной процесс)

1. **Покупатель** подходит к **кассовому аппарату** POS-системы с выбранными **товарами**.
3. **Кассир** открывает новую **продажу**.
4. Кассир вводит идентификатор товара.
5. Система записывает наименование товара и выдает его **описание**, цену и общую стоимостью Цена вычисляется на основе набора правил.

Кассир повторяет действия, описанные в пп. 3-4, для каждого наименования товара.

6. Система вычисляет общую стоимость покупки с налогом.
7. Кассир сообщает покупателю общую стоимость и предлагает оплатить покупку.
8. Покупатель оплачивает покупку, система обрабатывает **платеж.**
9. Система регистрирует продажу и отправляет информацию о ней внешней бухгалтерской системе (для обновления бухгалтерских документов и начисления комиссионных) и системе складского учета (для обновления данных).
10. Система выдает товарный чек.
11. Покупатель покидает магазин с чеком и товарами (если он что-то купил).

Здесь добавлен такой класс, как Каталог описаний, как необходимая категория – контейнер элементов – описаний, а также Магазин, как контейнер товаров и, наоборот, Строка продажи как абстракция элемента списка, которым является продажа. (Ledger – убрать !!!). Наоборот, не добавлен Чек, так как это **отчет** по другим классам (Продажа + Платеж).



Модель предметной области POS (из Лармана – стр. 178).

Темы 7-8 Быстрое проектирование, базовые принципы OOD и архитектурные шаблоны.

3. Быстрое проектирование. Базовые принципы объектного проектирования.
4. Знакомство с архитектурными паттернами. Слои и уровни. Декомпозиция приложения.

(Неформальное введение с минимумом деталей и небольшим числом простых примеров)

1. Быстрое проектирование. Базовые принципы объектного проектирования

(Упомянуть про паралич анализа и паралич проектирования.)

Что нужно делать после анализа требований и предметной области ? Как выполнять проектирование и переходить к реализации ? Можно двигаться по-разному. Ларман и UP рекомендуют заняться разработкой системных операций, реализующих выбранные для реализации прецеденты (Какие прецеденты выбирать – см. раздел планирования !). Для этого могут подойти диаграммы классов и **последовательностей** UML (наибольший упор делается именно на эти диаграммы!) и широкое использование **паттернов**. Далее диаграммы доводятся до такой степени детальности, что получение кода по ним – дело техники (см. книгу Лармана). При этом, конечно, нужно писать тесты и пр. Проектирование по Б. Страуструпу – см. выше. (+ книга В.В. Костерина и В.А. Камаева).

Несколько иной подход, хотя и близкий к нему, предлагает XP – быстрый сеанс проектирования и приступаем к созданию кода (в том числе и через тесты). Проектирование там сводится к минимуму, хотя объем времени на проектирование разный и зависит от задачи. Далее все покажет код и TDD. При обоих подходах предполагается страховать себя от ошибок, используя паттерны проектирования. Тут главное - не увязнуть (паралич анализа и проектирования). Мое мнение – нужно, по крайней мере, определить концептуально, какую общую архитектуру мы выбираем, а потом ее реализовывать.

1.1 Как выполнить быстрое проектирование ? Что для этого нужно ?

Как выполнить быстрое проектирование ? Во-первых, нужно заметить, что проектирование можно условно разделить на проектирование “в большом” - условно архитектурное - и проектирование “в малом” - низкоуровневые проектные решения. При отсутствии большого опыта в OOA / OOD проектирование системы даже "в большом" может занять много времени и привести к не очень удачным решениям. Совсем без проекта тоже нельзя. Чтобы начать планирование и приступить к реализации системы на начальных итерациях нужно иметь хотя бы какой-то первоначальный эскиз системы, ее архитектурный эскиз **(Первое приближение, как в др. итерационных методах)**. Как его получить? Чтобы не допустить каких-то глобальных ошибок (или – свести их к минимуму) нужно пользоваться шаблонами, а чтобы научиться пользоваться шаблонами, нужно познакомиться с базовыми принципами и приемами OOD. Эти принципы, а более подробно – паттерны, позволяют избежать признаков плохого проекта. Но главное – помнить, что окончательным воплощением проекта является его код (Джек Ривз – "Что такое программный проект").

1.2 Признаки плохого проекта (раскрыть – Мартин, стр. 151) :

- закрепощенность (эффект снежного кома при изменениях);
- неустойчивость (изменения в одних местах приводят к разрушению других);
- неподвижность (отсутствие четких компонент, которые можно повторно использовать);
- вязкость (сделать что-то правильно сложнее, чем неправильно);
- неоправданная сложность (сложная и бесполезная инфраструктура);
- неоправданные повторения (повт. куски);
- неопределенность (трудно читать и понимать).

1.3 Базовые принципы OOD (Роберт Мартин) :

12) SRP – Single Responsibility Principle
13) OCP – Open-Closed Principle
14) LSP – Liskov Substitution Principle
15) DIP – Dependency Inversion Principle
16) ISP – Interface Segregation Principle

1) Существует лишь одна причина, приводящая к изменению класса.

Примеры – модемная проблема (соединение и передача данных), графическое приложение (- потом ?!). Решение: разделение спаренных ответственностей и использование интерфейсов.

2) *Программные классы должны быть открыты для расширения и закрыты для модификации. (Бертран Майер)*

Примеры: Клиент – сервер.
Решение: использование абстрактных классов (интерфейсов) со множеством реализаций.
Паттерны Strategy и Template Method.
Метафоры : "Забрасывание удочек и первая пуля."

3) Подтипы (наследники) должны быть заменяемы их исходными типами.
(Барбара Лискоу).

Пример: прямоугольник и квадрат ()

```
class Rectangle
{
   public:
   void SetWidth(double w) {itsWidth=w;}
   void SetHeight(double h) {itsHeight=w;}
   double GetHeight() const {return itsHeight;}
   double GetWidth() const {return itsWidth;}
   private:
   double itsWidth;
   double itsHeight;
};
```

```
У наследника :
void Square::SetWidth(double w)
{
   Rectangle::SetWidth(w);
   Rectangle::SetHeight(w);
}
void Square::SetHeight(double h)
{
   Rectangle::SetHeight(h);
   Rectangle::SetWidth(h);
}
```

Однако :

```
void f(Rectangle& r)
{
   r.SetWidth(32); // calls Rectangle::SetWidth
}
```

Квадрат будет испорчен ! Выход – объявить эти методы в классе прямоугольник виртуальными ! Но это потребует изменения базового класса !!!

А если такой пример : (ASSERT() !)

Решение: использовать DBC – методику (Design By Contract) – Б. Мейер.
Правило Мейера + Модульные тесты + CRC (Class – Responsibility -Cooperation).

4) Принцип инверсии зависимостей (DIP) – в этой лекции более подробно

Модули высокого уровня не должны зависеть от модулей низкого уровня. Оба типа модулей обязаны зависеть от абстракций. Абстракции не должны зависеть от подробностей (реализации). Подробностям следует зависеть от абстракций.

5) *Клиенты не должны попадать в зависимость от методов, которыми они не пользуются.*
Речь идет о борьбе с "тучными" классами, которые взваливают на себя все и вся. Пример : Время и дверь. (Потом ?!)
Решение : Выделение нескольких интерфейсов и множественное наследование.

Итак, возвращаемся к 4 пункту…

Пример из Мартина :

2. Разбиение на слои
3. Абстрактные интерфейсы : картинки на стр. 208
3) Кнопка и лампа (стр. 211 – 212)

2. Понятие об архитектурном проектировании. Понятие о шаблонах (паттернах) проектирования. Знакомство с архитектурными паттернами. Концепция слоев приложения.

Собственно, именно это-то и нужно в данной лекции !
Ларман, стр. 439 :
"**Архитектура** – это набор важных решений, касающихся организации программной системы, выбора структурных элементов и их интерфейсов, затрагивающих поведение и взаимодействие этих элементов, их группировку в более крупные подсистемы и архитектурный стиль приложения" (из Буча и 3-х амиго, UML Users Guide)
Исследование и проектирование архитектуры в UP называют архитектурным анализом.

Архитектурный анализ и проектирование выполняется в ходе всей разработки, но основная его часть выполняется в фазе начала и особенно – развития проекта. Для анализа и выбора архитектуры особое внимание уделяют **нефункциональным требованиям в контексте функциональных** (таким как гибкость, адаптируемость, надежность, вопросы распределения системы, взаимосвязи с внешними программными и аппаратными компонентами и пр.) Выбор архитектуры влияет на планирование, поскольку добавляет ряд дополнительных технических задач в план. Разработку в фазе развития проекта необходимо вести “не вглубь, а вширь”, то есть – с учетом выбранной архитектуры реализовывать необходимый минимум прецедентов, но тем не менее опробовать архитектурный скелет проекта, на котором потом будет нарастать “мясо”.

Рассмотрим только **логическое представление** архитектуры. Принципы построения архитектуры нашли свое отражение в архитектурных шаблонах (паттернах). Наряду с ними в архитектурном анализе применяют и шаблоны других уровней.

Шаблон (паттерн) – именованное описание проблемы и ее решения, которые можно применить при разработке других систем. Это именованная пара "проблема / решение", содержащая рекомендации для применения в различных контекстах. Уровни шаблонов (проектирования) :

1. Архитектурные шаблоны (например, Layers, Service Layer и др.)

2. Шаблоны проектирования (например, GoF, GRASP, другие)
3. Идиомы – низкоуровневые шаблоны, ориентированные на конкретный язык или реализацию.

Кроме того, шаблоны разделяются по сферам использования и видам деятельности:

- шаблоны построения интерфейса;
- шаблоны проектирования и взаимодействия с БД;
- шаблоны построения корпоративных информационных приложений;

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

- шаблоны анализа;
- шаблоны тестирования;
- шаблоны планирования и пр.

Самый известный архитектурный шаблон – это шаблон **Layers** (Ларман)**.**
(Примеры архитектурных (и условно-архитектурных) шаблонов :

24)Шаблон Layers
25)Шаблон Facade
26)Шаблон Controller
27)Принцип Model-View Separation и Шаблон Observer (упомянуть).
28)Шаблон Информационный эксперт
29)Шаблоны Сильное зацепление и Слабое связывание
30) др. Принципы... Это очень спорно !!! - позже !!!)

3-х уровневая и многоуровневая архитектура. Шаблон Слои.

Принципы шаблона Слои (Ларман):

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.

Принцип 3-х уровневой архитектуры для информационных систем прост : Уровень представления, Уровень приложения и Уровень данных (технический слой, уровень или сама БД). При этом уровни четко разделены, зависимость – сверху вниз, в результате можно заменять источники данных, не трогая интерфейс, и наоборот, заменять интерфейс, не трогая средний уровень. Шаблон Layers несколько сложнее, он развивает эту концепцию дальше (больше слоев). Хорошо согласуется с этим шаблоном и развивает его принцип инверсии зависимостей (DIP), который говорит о том, что в ООП стараются при использовании слоев заменять зависимость от уровней зависимостями от интерфейсов, благодаря чему как-бы нижние уровни зависят от верхних в плане предоставляемой им функциональности (через интерфейс), а реальные зависимости "расшиты" интерфейсами (абстрактными классами).

Тема 9 Тестирование ПО. Модульное тестирование. Модульное тестрование в XP. Разработка через тестирование (TDD). Оболочка xUnit.

1. Роль тестирования в XP. Приемочные и модульные тесты. Нестандартный подход к тестированию и поиску ошибок.
2. Возражения критиков модульного тестирования.
3. Разработка через тестирование. Как это выглядит (концепция).
4. Как выглядит модульный тест. Как писать тесты ? (Примеры).
5. xUnit – приводим тесты к стандартному виду. CppUnit
6. Простейшие шаблоны тестирования (Подставной объект, Подделка, Триангуляция, Строка журнала и др.).

1. Роль тестирования в XP. Приемочные и модульные тесты. Нестандартный подход к тестированию.

Тестирование, как я уже упоминал, представляет собой второй (но не по важности) столп, одного из трех китов, на которых базируется XP. Не случайно ему посвящена целая книга Кента Бека. 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 процессах.

Основные роли (цели) тестирования в XP:

17) Поиск и исправление ошибок (средство отладки).
18) Приемка готового продукта (средство приемки).
19) Верификация проекта (устойчивость и целостность разработки)
20) Средство улучшения проекта (рефакторинга).
21) Документирование модулей (как CRC - карточки).
22) Сама разработка (Средство разработки в TDD).
23) и т.д.

(CRC – карточки : краткий комментарий.)

Основные виды тестов: 1) приемочные или функциональные тесты (разрабатываются в идеале заказчиком, либо тестировщиком); служат для приемки отдельного этапа работ, верификации проекта в целом; разрабатыаются на уровне прецедентов; лучше, когда они автоматизированы, в общем случае это просто набор входных данных, описание процедуры их ввода в систему, последовательность действий с системой и ожидаемый результат.

2) модульные тесты (разрабатываются самим разработчиком); цели – все перечисленные, кроме, может быть, приемки.

Так как проектирование, рефакторинг и внесение изменений происходят постоянно (это процесс, а не этап), то и тестирование – это перманентный процесс, который не выполняется на этапе отладки проекта, а выполняется всегда !!! Тестирование в XP носит всеобъемлющий характер. Это означает, что ни один модуль не считается разработанным, если к нему не прилагается набор тестов, полностью 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 главное : при использовании TDD тесты **являются инструментом разработки !**

Вкратце так : придумывая тесты, вы придумываете интерфейс и проектное решения

для вашей задачи. Главное здесь : **писать тесты модулей до того, как написан код самих модулей.**

2. Объем работ по модульному тестированию. Возражения критиков модульного тестирования (покороче – как в конспекте).

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". (Рассказать). 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.

Проект разрушается, становится похожим на тришкин кафтан. Это неизбежно. 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.

Не жалейте времени на тестирование – оно окупится с торицей !

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.

3) Тестирование не нужно

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.

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. Тестам несложно учиться, главное понять о попробовать концепцию.

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, поэтому сказав начальнику, что вы хотите добавить полное тестирование старого кода, вы скорее всего будете тестировать свое резюме, а не код.

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.

6) Бывает код, который нельзя протестировать

Проведя некоторое время в борьбе с оправданиями, чтобы не делать юнит тесты, мы должны покаяться. Это одно оправдание, которое имеет основу. Что могут сделать разработчики, если они общаются с железом, или внешней системой, которая не контролируется ими? Иногда эти внешние компоненты просто не будут нормально работать во время тестирования. Они могут быть слишком медленными, или возвращать непредсказуемые результаты при каждом вызове (зачастую с полным основанием - котировщик акций, выдающий одну и ту же цену при каждом вызове, облегчит тестирование, но посеет подозрения на бирже). Чтобы это преодолеть, разработчики могли бы написать тестовую оболочку, которая симулирует эти компоненты. Но для более сложных интерфейсов, вы должны взвесить затраты и отдачу. Можно держать пари, что NASA подробно тестирует свои программы и железо, но их бюджет, скорее всего, превышает ваш.

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.

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.

3. Разработка через тестирование (концепция)

Концептуально все умещается в фразу : "Модульные тесты пишутся до того, как пишется сам код". На самом деле, все выглядит примерно так : 1) Тесты, 2) Код, 3) Тесты проходят, 4) Рефакторинг, 5) Новые тесты не проходят (ломаем), 6) Код, 7) Тесты проходят и т.п. (Поломать, починить, улучшить). Улучшить : устранить дублирование, трудночитаемый и понимаемый код.

Но тесты сначала ! Как это может быть ? Да вот так. Вы сознательно идете по пути: желтый свет, красный свет, зеленый. Потом опять. (Метафора светофора). 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, вы таким образом, с помощью теста формируете интерфейс, затем – реализацию интерфейса, затем – улучшаете код (проект) 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.

Можно ли тестировать готовый код – можно, конечно (см. ранее) !

2. Как выглядит модульный тест. Как писать тесты ? (Примеры)
(все-таки, лучше это перед разработкой через тестирование).

1. Пример (просто тест).

Ну, например, тестируем простой класс, который считает величину гипотенузы прямоуг. треугольника (и не только). (Плохой пример !)

```
class calcRect3Angle {
     double a, b;
public :
     calcRect3Angle (double _a, _b)  :  a(_a), b(_b) {};
     double calcC () {
           return  sqrt(a*a + b*b);
      };
      void setAB(double _a, _b) { a = _a; b = _b;};
      double getA() {return a;};
      double getB() {return b;};
}

void testCalc() {     // Только Calc();
    calcRect3Angle t(4, 3);

    if (t.getA() = = 4) {
         cout << "OK!";
    }
    else {
         cout << "Error a != 4";
    }

    if (t.getB() = = 3) {
         cout << "OK!";
    }
    else {
         cout << "Error b != 3";
    }

    if (t.calcC() = = 5) {
         cout << "OK!";
    }
    else {
         cout << "Error :  c != 5";
    }

    t.setAB(5, 12);
```

```
    if (t.calcC() = = 13) {
        cout << "OK!";
    }
    else {
        cout << "Error :  c != 13";
    }
}
```

Потом мы решаем, что нам нужно возвращать без вызова calcC. Сначала мы делаем так :

```
double c;
double getC() {return c;}
calcC ( c = …; return c;)
```

Все хорошо, и тест проходит … Но нужно его сломать:

```
(t.getC() = = 5); (Перед вызовом calcC ! )
(t.getC() = = 13); (Перед вызовом calcC ! )

testSetGet() {
    calcRect3Angle t(4, 3);
    // t.getC() = = 5;
    t.setAB(5, 12);
    t.getC() = = 13;
}
```

2. Пример

В качестве примера можно использовать пример с классом Date – (как у ОИС) из книги Брюса Эккела Thinking in C++ :

Первый тест :

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

        int main() {
                Date mybday(1951, 10, 1);
                test(mybday.getYear() == 1951);
                test(mybday.getMonth() == 10);
                test(mybday.getDay() == 1);
                cout << "Passed: " << nPass << ", Failed: "  << nFail << endl;
        }
        /* Expected output: Passed: 3, Failed: 0 */
```

Второй тест :

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

int main() {
  Date mybday(1951, 10, 1);
  Date today;
   Date myevebday("19510930");
```

```
  // Test the operators
  test(mybday < today);
  test(mybday <= today);
  test(mybday != today);
  test(mybday == mybday);
  test(mybday >= mybday);
  test(mybday <= mybday);
  test(myevebday < mybday);
  test(mybday > myevebday);
  test(mybday >= myevebday);
  test(mybday != myevebday);

  // Test the functions
  test(mybday.getYear() == 1951);
  test(mybday.getMonth() == 10);
  test(mybday.getDay() == 1);
  test(myevebday.getYear() == 1951);
  test(myevebday.getMonth() == 9);
  test(myevebday.getDay() == 30);
  test(mybday.toString() == "19511001");
  test(myevebday.toString() == "19510930");

  // Test duration
  Date d2(2003, 7, 4);
  Date::Duration dur = duration(mybday, d2);
  test(dur.years == 51);
  test(dur.months == 9);
  test(dur.days == 3);

  // Report results:
  cout << "Passed: " << nPass << ", Failed: "
       << nFail << endl;
} ///:~
```

3. Пример (лучше вставить после упоминания изолированных тестов и подставных объектов)

Это пример из книги Мартина (не совсем пример теста, а пример подхода)

Допустим, нужно протестировать следующую систему (диаграмму) :

...

```
public void testPayroll () {
        MockDataBase* mdb = new MockDataBase();
        MockCheckWriter* mcw = new MockCheckWriter();
        Payroll* p = new Payroll(mdb, mcw);
        p.PayEmployees();
       assert(w.writtenCorrectly());
       assert(d.dataPostedCorrectly());
}
```

Диаграмма изоляции класса Payroll :

...

3. Простые шаблоны тестирования

Fake It, Three Angle, Mock Object, Log String, etc.
Подделка – подставляется код, который просто обеспечивает срабатывание (например, константа) , а потом – другие шаблоны (например, триангуляция), чтобы он перестал работать, и обобщение приводит к решению.
Триангуляция - написать два теста, которые совместно не работают, чтобы можно было выполнить обобщение. (Пример – тот же треугольник !)
Подставной объект – для изоляции тестируемого модуля от других модулей, с которыми он взаимодействует (например, источник данных).
Строка тестирования (отчета) – для отслеживания внутренних событий внутри модуля или сообщений, которые посылают друг другу модули при тестировании их совместного поведения (строка или целый лог - файл)**.** Пример – тот же треугольник. Нам нужно определить, было ли обращение к методу calcC ();

6. Каталог шаблонов тестирования :

Шаблоны TDD :
Test (Тесты нужны для всего, они должны быть автоматическими), **Isolated Test** (тесты должны быть изолирорванными друг от друга, изолируются также и тестируемые модули)**, Test List** (планирование 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).

Красной полосы (чтобы ее получить – то есть когда каие 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 чему-то учит, иногда это тест приложения...)**, Explanation Test** (приводите примеры в фоме 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)**, Break** (перерыв – вода :))**, Do over** (если ничего не помогает)**, Cheap Desk, Nice Chair** (хорошие кресла)**.**

Общие шаблоны тестирования :
Child Test (мелкие шаги – под-тест)**, Mock Object** (поддельный объект для изоляции)**, Self Shunt** (тестовый метод вместо объекта, с которым взаимодействует тестируемый объект)**, Log String** (Строка журнала)**, Broken Test** (Если вы программируете один, лучше оставлять последний тест сломанным)**, Clean Check-In** (Если в группе – все тесты должны работать)**.**

Зеленой полосы (чтобы ее получить): Fake It (Подделка)**, Triangulate** (Триангуляция)**, Obvious Implementation** (Очевидная реализация)**, One to Many** (От одного ко многим)**. И т.д.**

7. xUnit

Стандартная оболочка для тестирования (сущ. для разных языков)

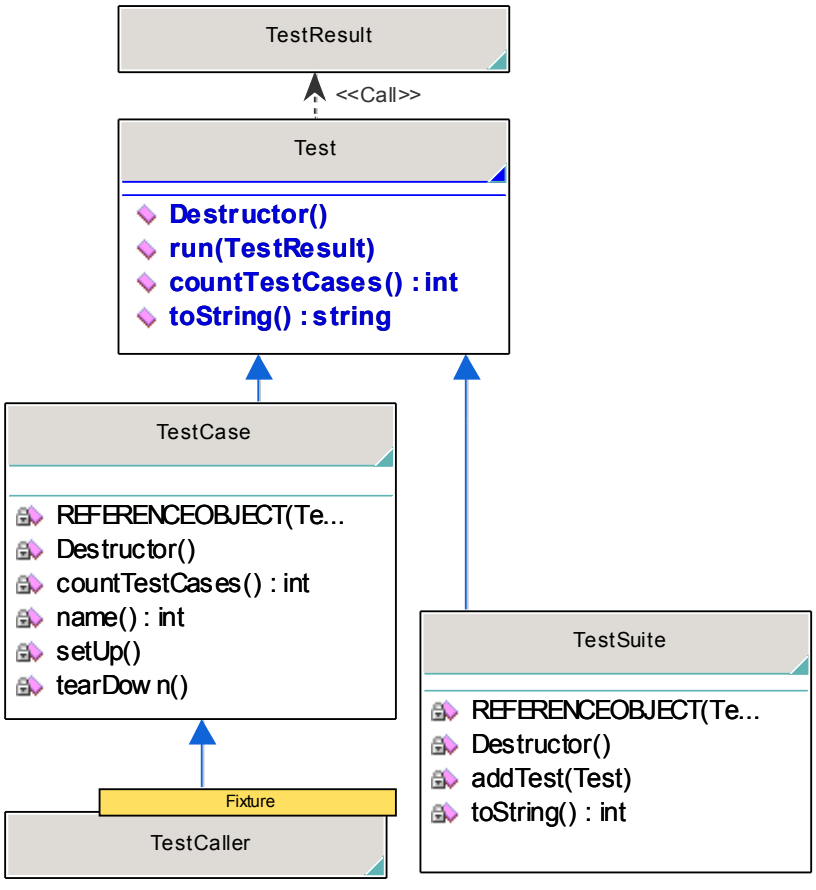
Assertion – логические выражения**, TestSuite, TestCase, TestMethode, Fixture.**

Набор тестов, тестовый случай, TestRunner, тестовый метод, фикстура.
- все через assert(), assertEquals().
- все тесты изолированы, все прогоняются.
- тест исключения – перехватывать.
- все тесты (список) - suite

Отчет : сколько запущено, сколько ошибок, сколько сбоев.

Структура оболочки (диаграмма классов) – по Фаулеру :

...



Добавить еще ассоциацию с базовым классом Test (шаблон Комозит) Наши тесты наследуют от TestCase. Он содержит метод suite() возвращающий Test (на самом деле – TestSuite)

Тема 10 Рефакторинг – продолжение … .

План:

- **Итак, роль (назначение) рефакторинга.**
- **Каталог шаблонов рефакторинга по Фаулеру (основные разделы, примеры шаблонов из этих разделов и примеры применения (короткие, даже ультракороткие !)**
- **Пример TDD (числа Фибоначчи) с указанием, где там был рефакторинг.**
- **Пример рефакторинга (чуть побольше) – простые числа (?!)**

1. Итак, роль (назначение) рефакторинга:

1.1. Устранение дублирования при запуске.
1.2. Облегчение модификации.
1.3. Сделать код более понятным.

Несколько общих советов от К.Бека (шаблонов) (В начало !):
- Согласование различий с последующим выделением метода и делегированием (основной подход к устранению дублирования);
- Изоляция изменений (!).
- Миграция данных – для переноса методов и атрибутов из класса в класс.

31)Числа Фибоначчи (К.Бек) – начать с него ?!! – Пример разработки через тестирование с элементами рефакторинга (минимальный пример).

а)
```
public void testFibonacci() {
  assertEquals(0, fib(0));
}
// Шаблон Подделка / Тестовые данные
int fib(int n) {
  return 0;
}
```

б)
```
public void testFibonacci() {
  assertEquals(0, fib(0));
  assertEquals(0, fib(1));
}
// Использование того же  метода – это пример исп. рефакторинга  Inline Method
// А теперь исп. шаблон тестирования  Триангуляция !
int fib(int n) {
  if (n = = 0) return 0;
  return 1;
}
```

в) Новый тест – опять триангуляция + шаблон рефакторинга Замещение алгоритма для устранения дублирования, примененный к тесту !
```
public void testFibonacci() {
  int cases[][] = {{0,0},{1,1}};
  for (int I = 0; I < cases.lenght; i++)
       assertEquals(cases[i][1], fib(cases[i][0]));
}
public void testFibonacci() {
```

```
  int cases[][] = {{0,0},{1,1}, {2,1}};   // int cases[2][] = {{0,0},{1,1}, {2,1}};
  for (int i = 0; i < cases.lenght; i++)    // sizeof(cases) / sizeof(cases[0]);
       assertEquals(cases[i][1], fib(cases[i][0]));
}
```

г) еще один тест (шаблон Тест Одного Шага):

```
public void testFibonacci() {
  int cases[][] = {{0,0},{1,1}, {2,1}, {3,2}};   // int cases[2][] = {{0,0},{1,1}, {2,1}};
  for (int i = 0; i < cases.lenght; i++)    // sizeof(cases) / sizeof(cases[0]);
       assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Меняем код модуля:

```
int fib(int n) {
  if (n = = 0) return 0;
  if (n < = 2) return 1;
  return 2;
}
```

// Теперь обобщаем код, пользуясь шаблоном тестирования – рефакторинга (КАКОЙ ШАБЛОН ?!).

```
int fib(int n) {
  if (n = = 0) return 0;
  if (n < = 2) return 1;
  return 1 + 1;
}
```

```
 int fib(int n) {
  if (n = = 0) return 0;
  if (n < = 2) return 1;
  return fib(n-1) + 1;
}
return fib(n-1) + fib(n-2);
```

Меняем условие - рефакторинг !!! (какой шаблон ?!)

```
if(n = = 1)  return 1;
```

32)Простые числа (Р.Мартин) :

Текст до рефакторинга :

```
import java.util.*;

public class GeneratePrimes
{
  /**
   * @param maxValue is the generation limit.
   */
  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue >= 2) // the only valid case
    {
      // declarations
      int s = maxValue + 1; // size of array
      boolean[] f = new boolean[s];
      int i;

      // initialize array to true.
```

```
      for (i = 0; i < s; i++)
        f[i] = true;

      // get rid of known non-primes
      f[0] = f[1] = false;

      // sieve
      int j;
      for (i = 2; i < Math.sqrt(s) + 1; i++)
      {
        for (j = 2 * i; j < s; j += i)
          f[j] = false; // multiple is not prime
      }

      // how many primes are there?
      int count = 0;
      for (i = 0; i < s; i++)
      {
        if (f[i])
          count++; // bump count.
      }

      int[] primes = new int[count];

      // move the primes into the result
      for (i = 0, j = 0; i < s; i++)
      {
        if (f[i])             // if prime
          primes[j++] = i;
      }

      return primes;  // return the primes
    }
    else // maxValue < 2
      return new int[0]; // return null array if bad input.
  }
}
```

Тесты :

```
import junit.framework.*;
import java.util.*;

public class TestGeneratePrimes extends TestCase
{
  public TestGeneratePrimes(String name)
  {
    super(name);
  }

  public void testPrimes()
  {
    int[] nullArray = GeneratePrimes.generatePrimes(0);
    assertEquals(nullArray.length, 0);

    int[] minArray = GeneratePrimes.generatePrimes(2);
    assertEquals(minArray.length, 1);
    assertEquals(minArray[0], 2);

    int[] threeArray = GeneratePrimes.generatePrimes(3);
    assertEquals(threeArray.length, 2);
    assertEquals(threeArray[0], 2);
    assertEquals(threeArray[1], 3);

    int[] centArray = GeneratePrimes.generatePrimes(100);
    assertEquals(centArray.length, 25);
    assertEquals(centArray[24], 97);
  }
```

```
}
```

Текст после рефакторинга :

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum.  The algorithm used is the Sieve of Eratosthenes.
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out all its
 * multiples.  Repeat until there are no more multiples
 * in the array.
 */

public class PrimeGenerator
{
  private static boolean[] crossedOut;
  private static int[] result;

  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      uncrossIntegersUpTo(maxValue);
      crossOutMultiples();
      putUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void uncrossIntegersUpTo(int maxValue)
  {
    crossedOut = new boolean[maxValue + 1];
    for (int i = 2; i < crossedOut.length; i++)
      crossedOut[i] = false;
  }

  private static void crossOutMultiples()
  {
    int limit = determineIterationLimit();
    for (int i = 2; i <= limit; i++)
      if (notCrossed(i))
        crossOutMultiplesOf(i);
  }

  private static int determineIterationLimit()
  {
    // Every multiple in the array has a prime factor that
    // is less than or equal to the root of the array size,
    // so we don't have to cross of multiples of numbers
    // larger than that root.
    double iterationLimit = Math.sqrt(crossedOut.length);
    return (int) iterationLimit;
  }

  private static void crossOutMultiplesOf(int i)
  {
    for (int multiple = 2*i;
         multiple < crossedOut.length;
         multiple += i)
      crossedOut[multiple] = true;
  }

  private static boolean notCrossed(int i)
  {
    return crossedOut[i] == false;
  }
```

```
  private static void putUncrossedIntegersIntoResult()
  {
    result = new int[numberOfUncrossedIntegers()];
    for (int j = 0, i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        result[j++] = i;
  }

  private static int numberOfUncrossedIntegers()
  {
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        count++;

    return count;
  }
}

import junit.framework.*;
public class TestGeneratePrimes extends TestCase
{
  public static void main(String args[])
  {
      junit.swingui.TestRunner.main(
        new String[] {"TestGeneratePrimes"});
  }
  public TestGeneratePrimes(String name)
  {
    super(name);
  }

  public void testPrimes()
  {
    int[] nullArray = PrimeGenerator.generatePrimes(0);
    assertEquals(nullArray.length, 0);

    int[] minArray = PrimeGenerator.generatePrimes(2);
    assertEquals(minArray.length, 1);
    assertEquals(minArray[0], 2);

    int[] threeArray = PrimeGenerator.generatePrimes(3);
    assertEquals(threeArray.length, 2);
    assertEquals(threeArray[0], 2);
    assertEquals(threeArray[1], 3);

    int[] centArray = PrimeGenerator.generatePrimes(100);
    assertEquals(centArray.length, 25);
    assertEquals(centArray[24], 97);
  }

  public void testExhaustive()
  {
    for (int i = 2; i<500; i++)
        verifyPrimeList(PrimeGenerator.generatePrimes(i));
  }

  private void verifyPrimeList(int[] list)
  {
    for (int i=0; i<list.length; i++)
        verifyPrime(list[i]);
  }

  private void verifyPrime(int n)
  {
    for (int factor=2; factor<n; factor++)
      assert(n%factor != 0);
  }
```

```
}
```

Какие шаблоны рефакторинга применены :

2. Выделение метода
3. Переименование метода / поля
4. Декомпозиция условия
5. Поясняющие переменные
    5) Шаблоны тестирования (Еще один тест + Поясняющий тест)

4. Список рефакторингов по М. Фаулеру:

1. Составление 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)

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. Организация 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)
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
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)

7. Крупные рефакторинги
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)

5. Краткое описание некоторых рефакторингов:

5.1. Составление методов

5.1.1. Выделение метода (Extract Method)

Проблема: *Имеется фрагмент кода, который может быть сгруппирован.*
Решение: Преобразовать фрагмент в метод, имя которого объясняет его назначение.
Пример кода:
До:

```
void printOwing() {
  printBanner();
  //print details
  System.out.println ("name:     " + _name);
  System.out.println ("amount  " + getOutstanding());
 }
```

После:

```
void printOwing() {
        printBanner();
        printDetails(getOutstanding());
}
void printDetails (double outstanding) {
        System.out.println ("name:     " + _name);
        System.out.println ("amount    " + outstanding);
}
```

5.1.2. Встраивание метода (Inline Method)

Проблема: *Содержание метода очевидно следует из его названия*
Решение: Встроить тело метода в вызывающие его модули и удалить метод
Пример кода:
До:

```
int getRating() {
        return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
        return _numberOfLateDeliveries > 5;
}
```

После: int getRating() {

```
                return (_numberOfLateDeliveries > 5) ? 2 : 1;
        }
```

5.1.3. Введение поясняющей переменной (Explaining Variable)

Проблема: *Имеется сложное выражение.*

Решение: Поместить результат выражения или части выражения во временную переменную, название которой объясняет ее назначение.

Пример кода:

До :

```
if ((platform.toUpperCase().indexOf("MAC") > -1) &&
      (browser.toUpperCase().indexOf("IE") > -1) &&
       wasInitialized() && resize > 0 )
 {   // do something   }
```

После :

```
final boolean isMacOs     = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE")  > -1;
final boolean wasResized  = resize > 0;
if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
// do something
}
```

5.1.4. Замена временной переменной вызовом метода (Replace Temp with Query)

Проблема: *Вы используете временную переменную для хранения результата выражения*

Решение: Выделите выражение в метод. Замените все ссылки на данную переменную на вызовы метода. Новый метод может быть использован в новых методах.

Пример кода:

До :

```
                double basePrice = _quantity * _itemPrice;
                if (basePrice > 1000)
                        return basePrice * 0.95;
                else
                        return basePrice * 0.98;
```

После :

```
                if (basePrice() > 1000)
                        return basePrice() * 0.95;
                else
                        return basePrice() * 0.98;

                double basePrice() {
                        return _quantity * _itemPrice;
                }
```

5.1.5. Замена метода объектом метода (Replace Method with Method Object).

Проблема: *Имеется длинный метод, который использует локальные переменные таким образом, что нельзя применить Извлечение метода*

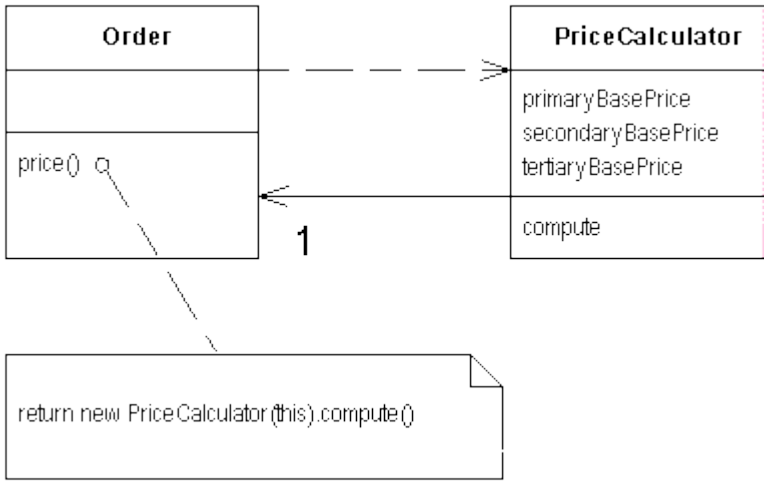
Решение: Преобразовать метод в объект таким образом, что все локальные переменные становятся полями этого объекта. Затем можно разделить методы на другие методы того же объекта.

Пример :

До :

```
class Order...
        double price() {
                double primaryBasePrice;
                double secondaryBasePrice;
                double tertiaryBasePrice;
                // long computation;
                ...
        }
```

После :



5.1.6. Замещение алгоритма (Substitute Algorithm)
Проблема: *Вы хотите заменить алгоритм другим (более понятным)*
Решение: Заменить тело метода другим алгоритмом (другой реализацией)
Пример :
До :

```
String foundPerson(String[] people){
        for (int i = 0; i < people.length; i++) {
                if (people[i].equals ("Don")){
                        return "Don";
                }
                if (people[i].equals ("John")){
                        return "John";
                }
                if (people[i].equals ("Kent")){
                        return "Kent";
                }
        }
        return "";
}
```

После :

```
String foundPerson(String[] people){
        List candidates = Arrays.asList(new String[] {"Don", "John",
                                "Kent"});
        for (int i=0; i<people.length; i++)
                if (candidates.contains(people[i]))
                        return people[i];
        return "";
}
```

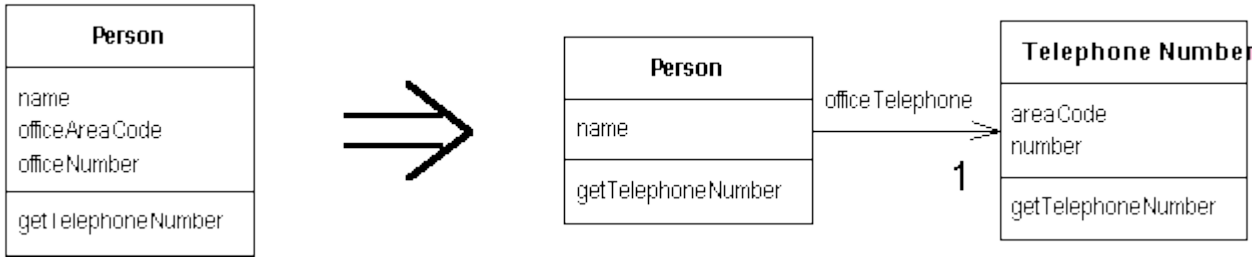
- **Перемещение функций между объектами**

5.2.1. Перемещение метода (Move Method)
Проблема: *Метод больше связан с другим классом, чем с тем, в котором он определен*
Решение: 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.

5.2.2. Перемещение поля (Move Field)
Проблема: *Поле (атрибут) больше связано с другим классом, чем с тем, в котором оно определено.*
Решение: Создать новое поле в целевом классе и изменить все связанные с ним

методы.

5.2.3. Выделение класса (Extract Class)
Проблема: *Имеется один класс, выполняющий работу двух.*
Решение: Создать новый класс и перенести необходимые методы и атрибуты из старого класса в новый.
Пример :
До : **После :**



5.2.4. Встраивание класса (Inline Class)
Проблема: *Класс имеет очень мало обязанностей.*
Решение: Перенести необходимые методы и атрибуты из класса в другой и удалить класс.

- **Организация данных**

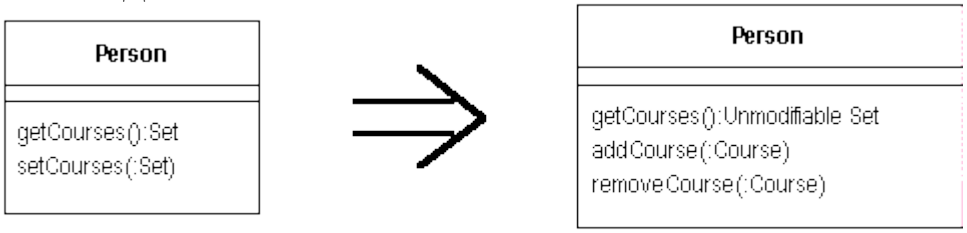
- **Инкапсуляция поля (Incapsulate field)**

Проблема: *Класс имеет общедоступное поле.*
Решение: Сделать поле частным (закрытым) и создать методы доступа.
Пример :
До :
```
public String _name
```
После :
```
private String _name;
public String getName() {return _name;}
public void setName(String arg) {_name = arg;}
```

5.3.2. Самоинкапсуляция поля (Self – Incapsulate Field)

5.3.3. Инкапсуляция коллекции (полезная вещь!) – Incapsulate Collection
Проблема: *Метод возвращает коллекцию.*
Решение: Обеспечить возврат коллекции только для чтения и создать методы вставки и удаления.
Пример :

До : **После :**

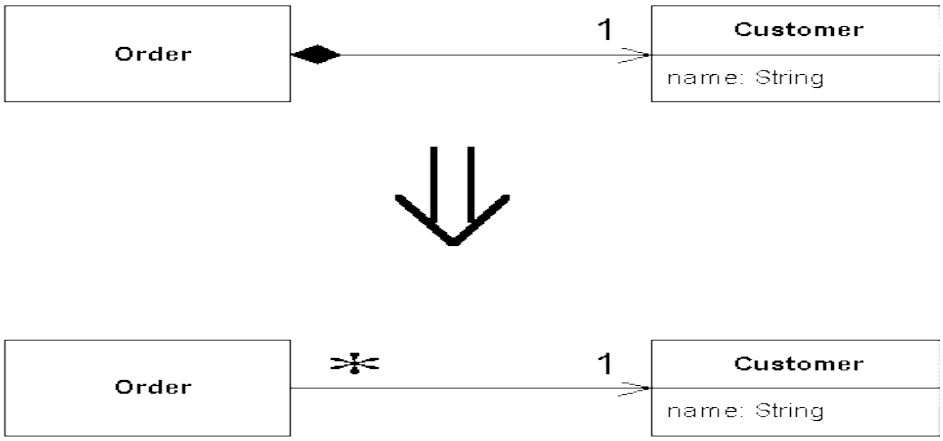


5.3.4. Замена значения ссылкой (Change Value to Reference)

Проблема: *Имеется много экземпляров одного и того же класса, которые необходимо заменить на один объект.*

Решение: Преобразовать объект в ссылку на объект.

Пример :

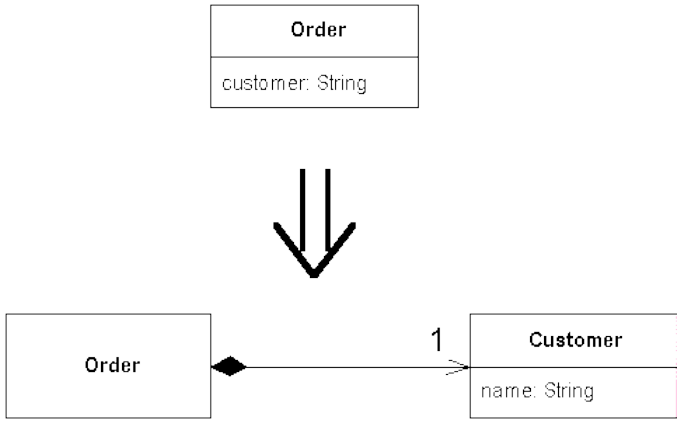


5.3.5. Замена массива / структуры / данных объектом (Replace … with Object).

Проблема: *Имеется элемент данных, который требует других данных или обладает связанным с ним поведением.*

Решение: Преобразовать элемент данных в объект.

Пример:

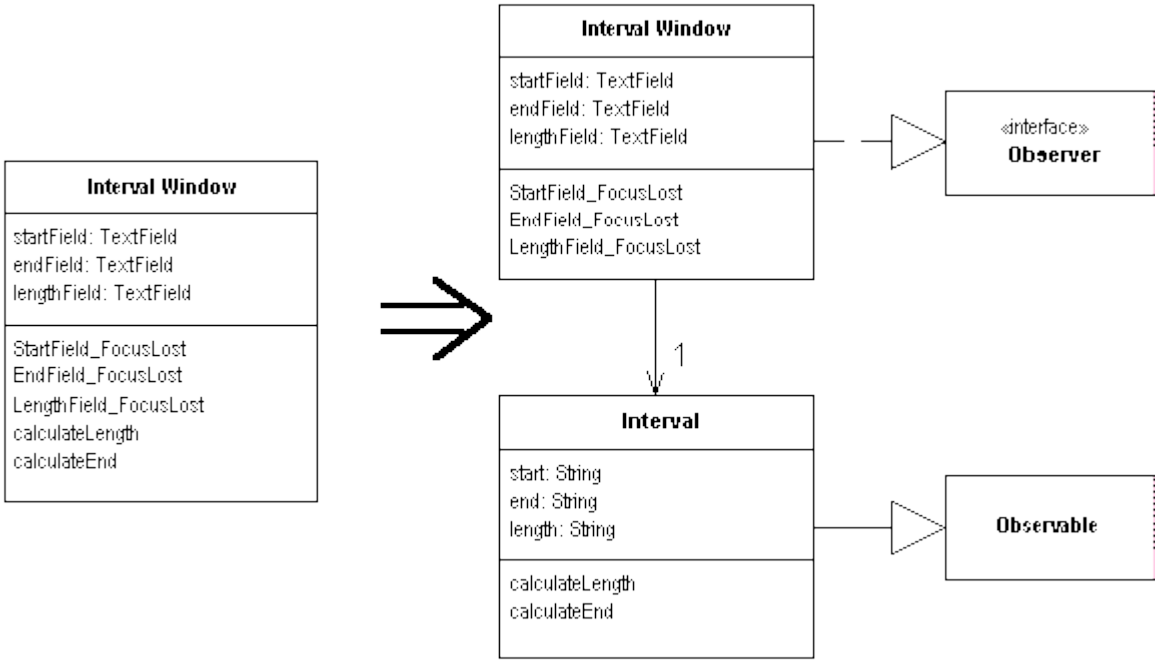


5.3.6. Дублирование видимых данных (Duplicate Observed Data) – для разделения классов GUI и предметной области.

Проблема: *Имеются данные предметной области, которые присутствуют только в пользовательском интерфейсе и для них нужен доступ из классов предметной области.*

Решение: Скопировать данные в объект предметной области. Создать объект-наблюдатель (использовать шаблон Наблюдатель) для синхронизации двух наборов данных.

Пример :



- **Упрощение условных выражений**

5.4.1. Декомпозиция условного оператора (Decompose Conditional)
Проблема: *Имеется сложный условный оператор (if-then-else).*
Решение: Извлечь методы из условной части, а также из обеих ветвей.
Пример :
До :

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

После :

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

5.4.2. Консолидация условного выражения (Consolidate Conditional Expression)
Проблема: *Имеется последовательность условий с одинаковым результатом.*
Решение: Объединить условия в одно условное выражение и извлечь его в отдельный метод.
Пример :
До :

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

После :

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

5.4.3. Консолидация дублирующихся условных фрагментов (Consolidate Duplicate Conditional Fragments)
Проблема:*Один и тот же фрагмент кода присутствует во всех ветвях условного оператора.*
Решение: Перенести повторяющийся фрагмент вовне условного оператора.

Пример :
До :

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

После:

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

5.4.4. Замена условного оператора на полиморфизм (Replace Conditional with Polymorphism).

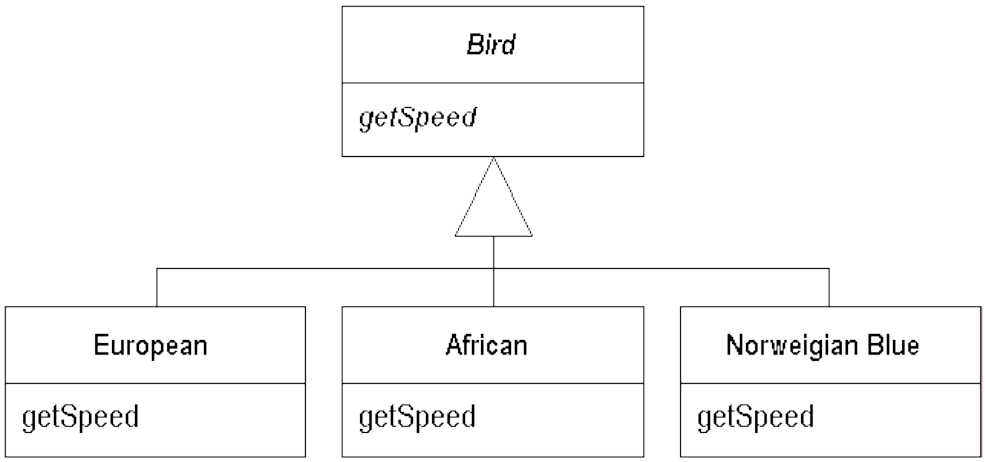
Проблема: *Имеется оператор выбора (или ветвление) который выбирает различное поведение, зависящее от типа объекта.*

Решение: Перенести различные ветви оператора в переопределяемый метод подкласса. Сделать исходный метод абстрактным.

Пример:
До:

```
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");
        }
```

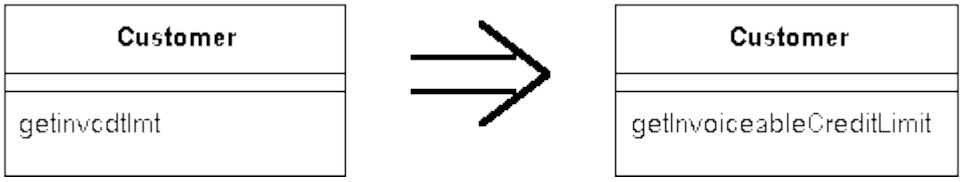
После:



- **Упрощение вызова методов**
  - **Переименование метода (Rename Method)**

Проблема: *Имя метода не раскрывает его назначение.*
Решение: Изменить имя метода.
Пример:

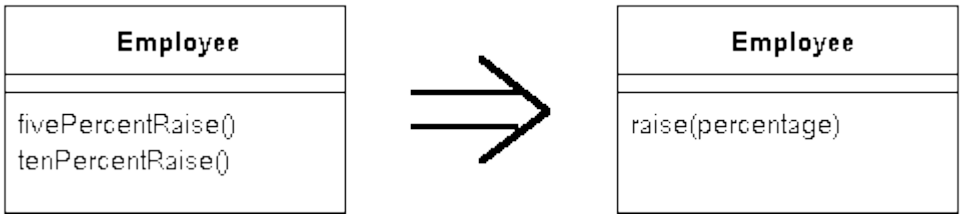
До : **После :**



5.5.2. Добавление и удаление параметра (Add / remove Parameter).
Проблема: *Метод нуждается в дополнительной информации от вызывающего объекта.*
Решение: Добавить параметр в метод.

5.5.3. Параметризация метода (Parameterize method).
Проблема: *Метод нуждается в дополнительной информации от вызывающего объекта.*
Решение: Добавить параметр в метод.
Пример:

До : **После :**



5.5.4. Замена конструктора фабричным методом (Replace Constructor With Factory Method)
Проблема: *Вы хотите делать что-то более сложное, чем простое создание при создании объекта. (А вернее сказать – есть необходимость спрятать детали создания объекта).*
Решение: Заменить конструктор фабричным методом (паттерн проектирования Фабрика).
Пример :
До :

```
Employee (int type) {
        _type = type;
}
```

После :

```
static Employee create(int type) {
        return new Employee(type);
}
```

5.5.5. Сокрытие метода (Hide Methode)
Проблема: *Метод класса не используется другими классами.*
Решение: Сделать метод закрытым.

- **Решение задач обобщения**

- **Подъем поля / метода (Pull-Up Filed / Method)**

Проблема: *Два подкласса имеют общее поле.*

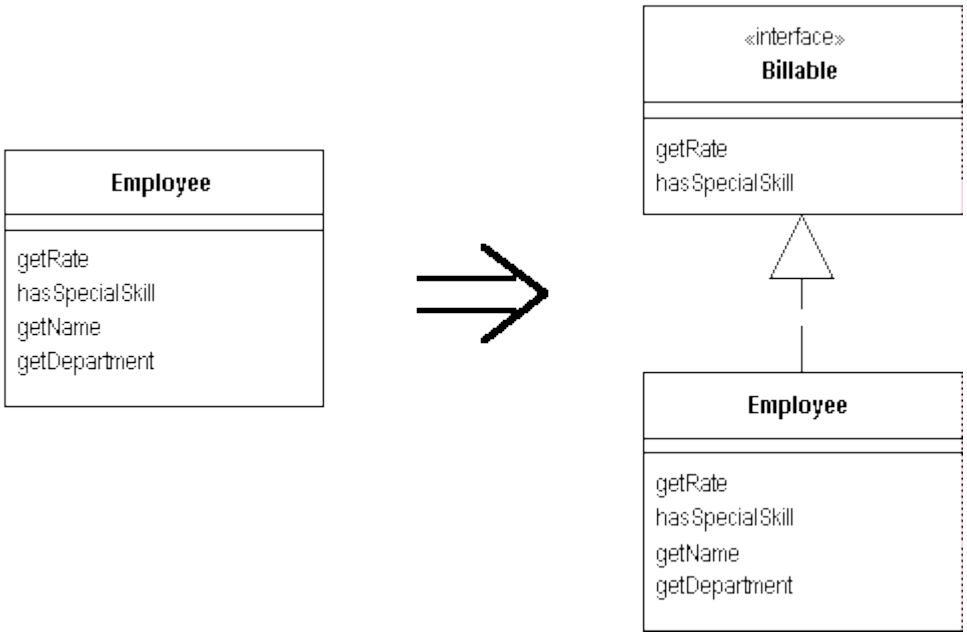
Решение: Переместить поле в родительский класс

5.6.2. Спуск поля / метода (Push Down Filed / Method) – аналогично…

5.6.3. Выделение подкласса (Extract Subclass)
Проблема: *Класс имеет свойства, которые используются только у каких-то конкретных экземпляров или у типа экземпляров.*
Решение: Создать подкласс для этой группы экземпляров.

5.6.4. Выделение родительского класса (Extract Superclass)
Проблема: *Имеются два класса с похожими свойствами.*
Решение: Создать родительский класс, выделив туда общие свойства этих классов.

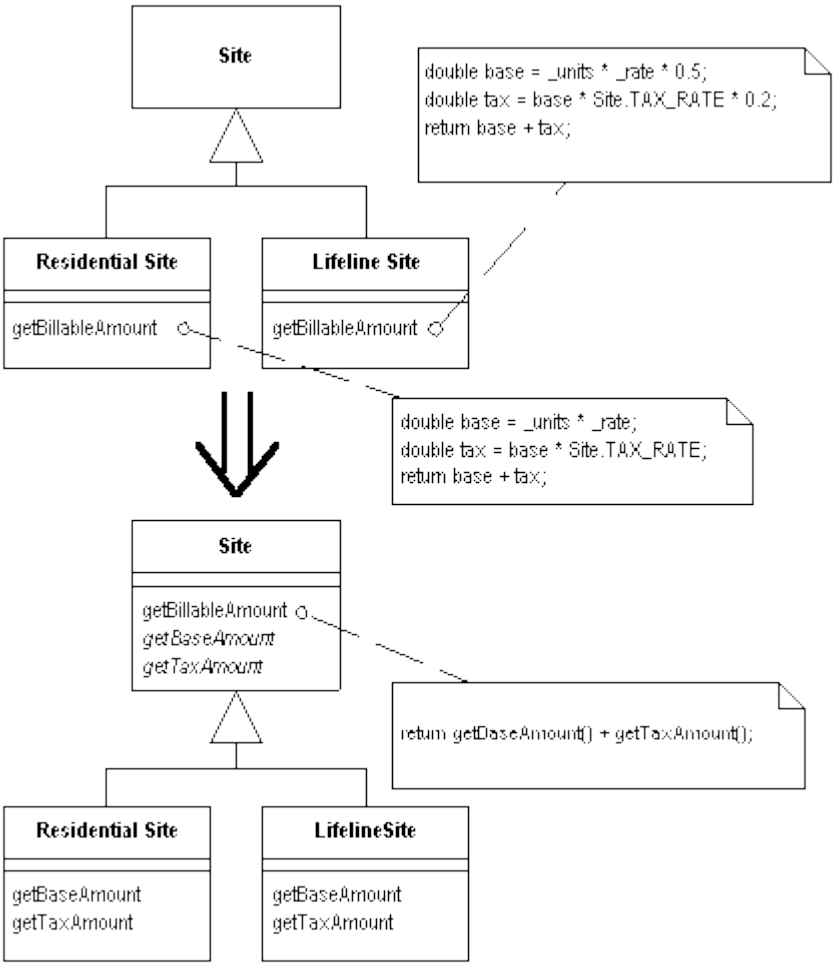
5.6.5. Выделение интерфейса (Extract Interface)
Проблема: *Несколько клиентов используют одно и то же подмножество интерфейса класса, или два класса имеют пересекающуюся часть интерфейса.*
Решение: Выделить это подмножество интерфейса в отдельный интерфейс.
Пример :



5.6.6. Свертывание иерархии (Collapse Hierarchy)
Проблема: *Родительский класс и класс наследника не слишком отличаются.*
Решение: Объединить классы в один.

5.6.7. Формирование шаблона метода (Form Template Method)
Проблема: *Имеются два метода в подклассах, которые выполняют похожие шаги в похожем порядке, хотя детали различны.*
Решение: Постепенно сделать эти методы одинаковыми, преобразуя шаги в методы с одинаковой сигнатурой. Затем можно выполнить Подъем метода.

Пример :



Замечание : на самом деле это проявление общего приема, не только для подклассов, но и вообще в случае дублирования методов : Согласовать различия (это тоже один из шаблонов), а затем – Вынести метод в отдельный метод, обеспечить удобный интерфейс и заменить тело дублирующихся методов вызовом нового метода. Особенность в данном шаблоне – использование наследования.

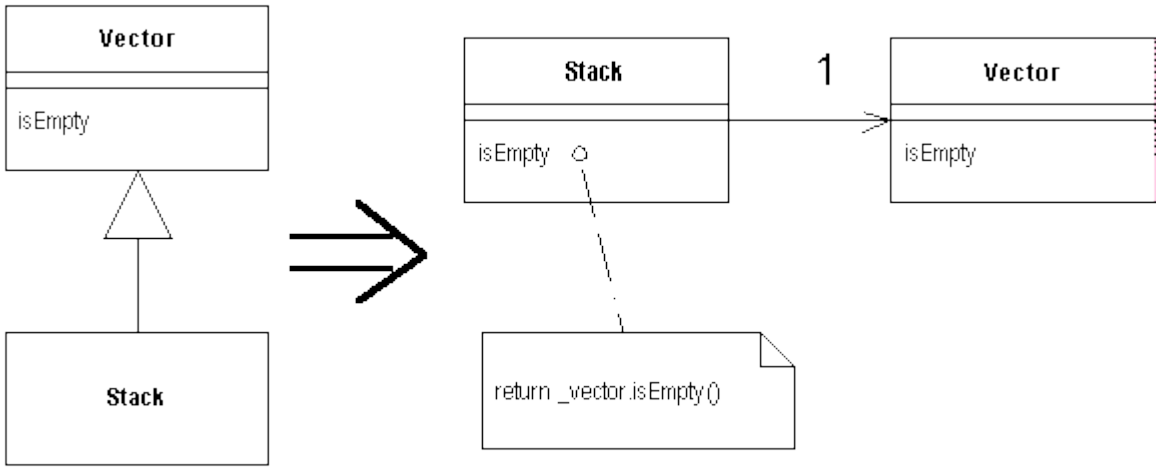
5.6.7. Замена наследования делегированием и наборот (Replace Inheritance with Delegation vice versa).

Проблема: *Наследник класса использует только часть интерфейса родителя или не хочет наследовать его данные***.**

Решение: Создать поле для родительского класса, преобразовать методы для выполнения делегирования этому классу и устранить наследование.

Пример:

До : **После :**



5.7. Крупные рефакторинги (рассмотрим потом в отдельной лекции !) :

5.7.1. Разделение наследования

5.7.2. Преобразование процедурного проекта в объекты

5.7.3. Отделение предметной области от представления

5.7.4. Выделение иерархии.

Источники неприятных запахов

Запах	Рефакторинги
Альтернативные классы с разными интерфейсами (Alternative Classes with Different Interfaces), стр. 97	Переименование метода *(Rename Method, 277),* Перемещение метода *(Move Method, 154)*
Комментарии (Comments), стр. 99	Выделение метода *(Extract Method, 124),* Введение утверждения *(Introduce Assertion, 270)*
Классы данных (Data Class), стр. 98	Перемещение метода *(Move Method, 154),* Самоинкапсуляция поля *(Self Encapsulate Field, 181),* Инкапсуляция коллекции *(Encapsulate Collection, 214)*
Группы данных (Data Clumps), стр. 92	Выделение класса *(Extract Class. 161),* Введение граничного объекта *(Introduce Parameter Object, 297),* Сохранение всего объекта *(Preserve Whole Object, 291)*
Расходящиеся модификации (Divergent Change), стр. 90	Выделение класса *(Extract Class, 161)*
Дублирование кода (Duplicated Code), стр. 86	Выделение метода *(Extract Method, 124),* Выделение класса *(Extract Class, 161),* Подъем метода *(Pull Up Method, 323),* Формирование шаблона метода *(Form Template Method. 344)*
Завистливые функции (Feature Envy, стр. 91	Перемещение метода *(Move Method, 154),* Перемещение поля *(Move Field, 158),* Выделение метода *(Extract Method. 124)*
Неуместная близость (Inappropriate Intimacy), стр. 96	Перемещение метода *(Move Method, 154),* Перемещение поля *(Move Field, 158),* Замена двунаправленной связи однонаправленной *(Change Bidirectional Association to Unidirectional, 207),* Замена наследования делегированием *(Replace Inheritance with Delegation, 352),* Сокрытие делегирования *(Hide Delegate, 168)*
Неполнота библиотечного класса (Incomplete Library Class), стр. 97	Введение внешнего метода *(Introduce Foreign Method, 172),* Введение локального расширения *(Introduce Local Extension, 174)*
Большой класс (Large Class), стр. 88	Выделение класса *(Extract Class, 161),* Выделение подкласса *(Extract Subclass, 330),* Выделение интерфейса *(Extract Interface, 341),* Замена значения данных объектом *(Replace Data Value with Object, 184)*
Ленивый класс (Lazy Class), стр. 94	Встраивание класса *(Inline Class. 165),* Свертывание иерархии *(Collapse Hierarchy, 343)*

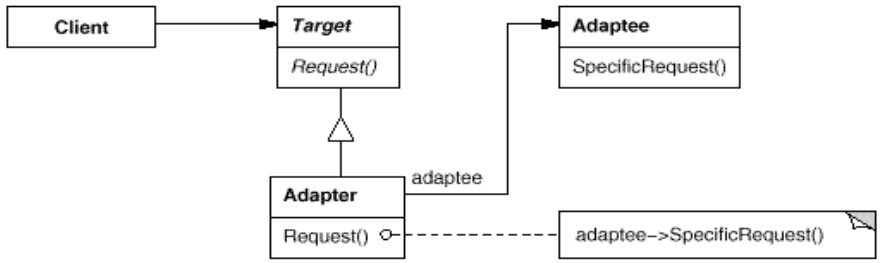
Тема 11 Очень краткое описание основных паттернов проектирования GoF

(Возможны ошибки и некорректные трактовки !)

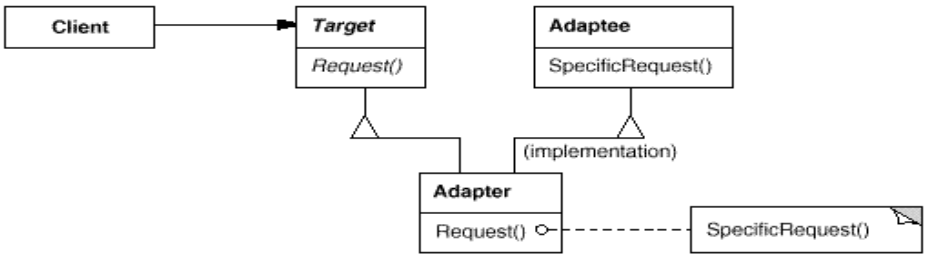
I. Паттерны структурирования :

1. Адаптер - Преобразует существующий интерфейс класса в другой интерфейс, который понятен клиентам.

Адаптер объектов :

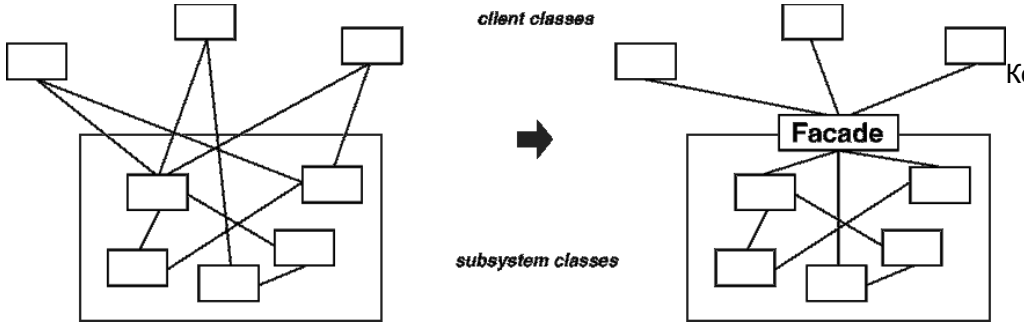


Адаптер классов :



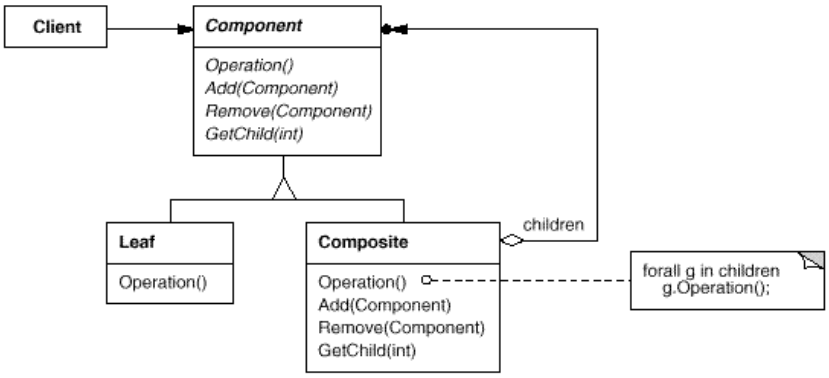
Участники : клиент, целевой объект (Target), адаптируемый объект (Adaptee), адаптер. В первом случае адаптер *делегирует* ассоциированному с ним адаптируемому объекту запрос клиента, который обращается к известному ему интерфейсу Target. Во втором случае адаптер *наследует от обоих классов* : и от целевого, и от адаптируемого объекта. Он может переопределять / дополнять поведение адаптируемого класса (но не его подклассов).

2. Фасад - Типичный структурный шаблон, цель которого – обеспечить удобный интерфейс к сложной иерархии классов, которую хотелось бы спрятать от клиентов (интерфейс к пакету, слою, уровню и т.п.)



Пример – Компилятор и др.

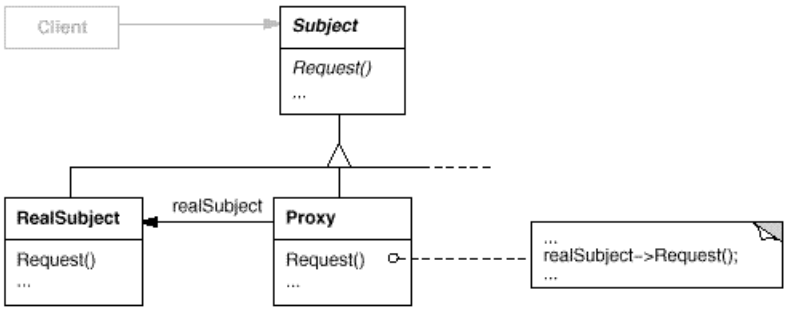
3. Композит - Один из часто используемых шаблонов : группирует объекты в иерархические структуры для представления отношений типа "часть-целое", что позволяет клиентам работать с единичными объектами так же, как с группами объектов.



Участники : Ключевой частью паттерна является абстрактный класс Component, который предназначен и для представления части, и для представления целого для унификации работы с ними. Композит состоит из компонентов, но и сам является им же. Это позволяет одинаково обрабатывать и листовые и иерархические объекты. Часто применяется с Итератором.

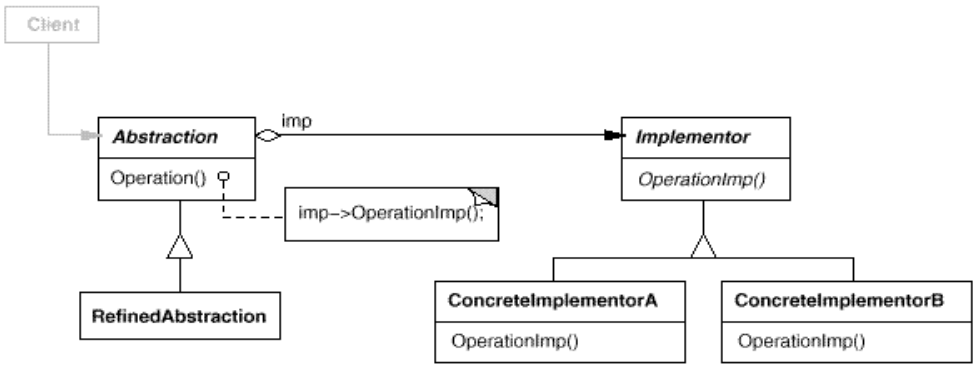
Пример : компьютер и комплектующие.

4. Заместитель (Proxy) - Замещает собой другой объект для контроля доступа к нему.



Участники : Subject - интерфейс замещаемого объекта, RealSubject - замещаемый объект, Proxy – Замещающий объект с тем же интерфейсом. Примеры использования : БД (Proxy замещает объект предметной области, но знает о способах его сохранения и загрузки из БД), доступ к Интернет и пр.

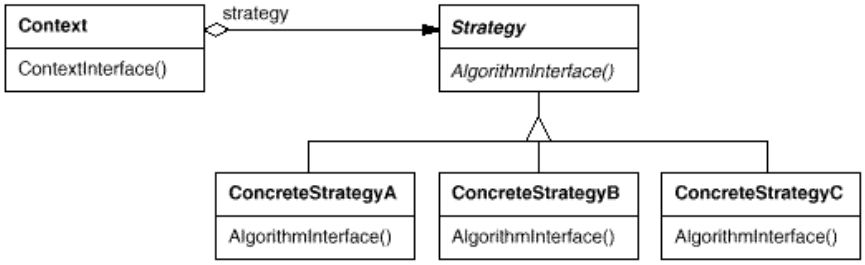
5. Мост - Отделяет абстракцию класса от его реализации, благодаря чему появляется возможность независимо изменять то и другое.



Участники: клиент, абстракция (интерфейс сущности), реализация (Implementor - к ней делегируется сообщение), конкретные реализации, наследники абстракции (расширяют ее интерфейс). Обычно адаптер применяется уже пост-фактум, а мост – в процессе проектирования. Мост обычно сочетается с Фабрикой… (Пример с окнами).

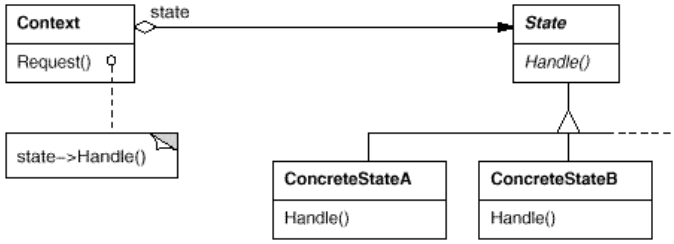
II. Паттерны поведения

6. Стратегия – Базовый паттерн поведения. Определяет множество алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого. При этом можно изменять алгоритм независимо от клиента, который им пользуется (Паттерн автоматически возникает при реализации рефакторинга Замена условной логики на полиморфизм).

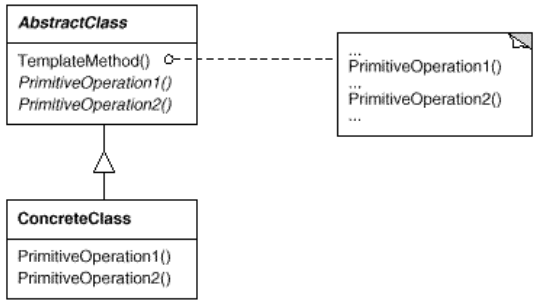


Участники : Контекст (куда помещается алгоритм), Стратегия (класс интерфейса алгоритма), ее конкретные реализации (варианты алгоритмов).

7. Состояние - паттерн для объектной реализации модели конечного автомата, в чем-то напоминает Стратегию (Можно сказать – частный случай Стратегии). Позволяет объекту менять свое поведение в зависимости от состояния. Для состояния определяется интерфейс, с которым работает клиент машины состояний, а конкретные состояния являются его реализациями.

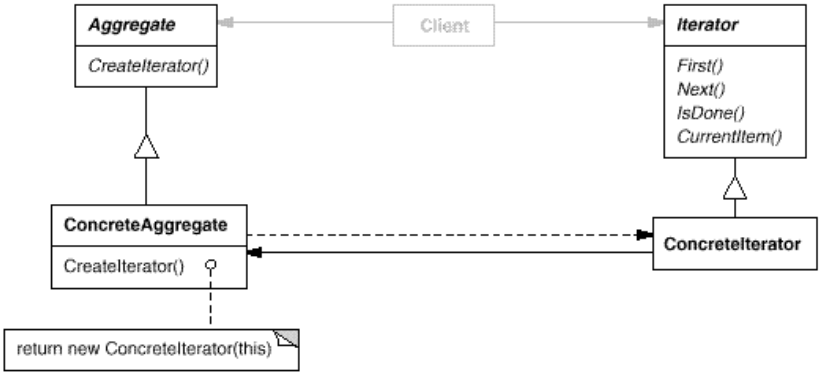


8. Шаблонный метод - Определяет структуру алгоритма, перераспределяя ответственность за некоторые его шаги на подклассы. При этом подклассы могут переопределять шаги алгоритма, не меняя его общей структуры.



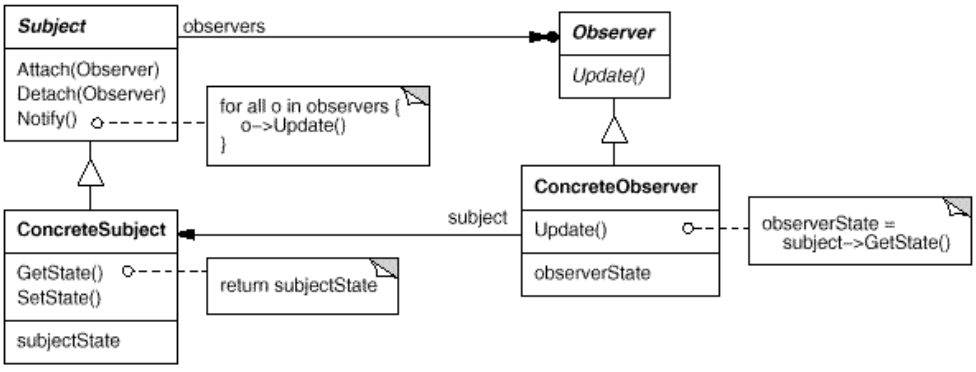
Участники : абстрактный класс, содержащий шаблонный метод, конкретный наследник, реализующий примитивные операции.

9. Итератор - Дает возможность последовательно перебрать все элементы составного объекта, не раскрывая его внутреннего представления (для инкапсуляции коллекций, композитов и пр). Часто используется с шаблонами Композит и Фабричный метод.



Участники : Клиент, использующий интерфейсы Итератора и Класса-агрегата, конкретные агрегаты и итераторы, реализующие интерфейсы.

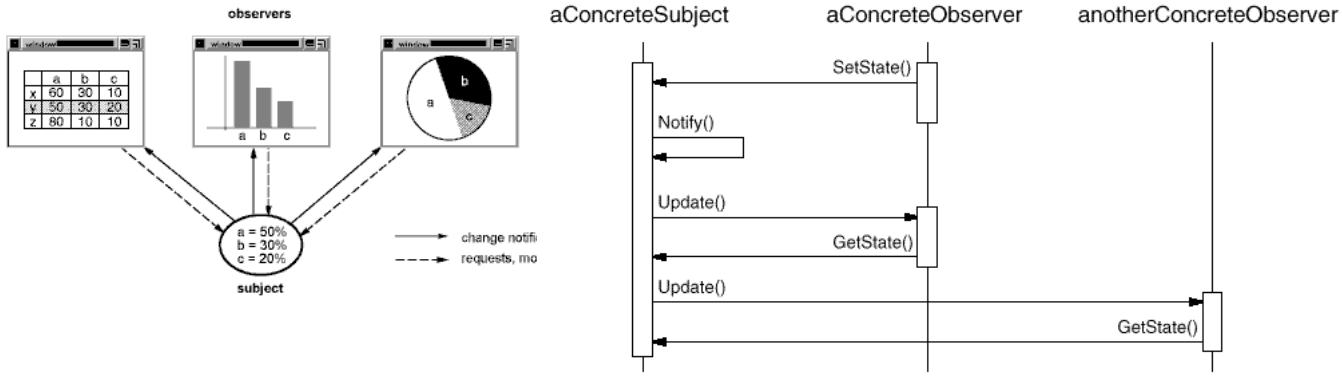
10. Наблюдатель - Специфицирует зависимость типа "один ко многим" между различными объектами, так что при изменении состояния одного объекта все зависящие от него получают извещение и автоматически обновляются. Паттерн имеет много применений, одно из которых – синхронизация данных между объектами уровня представления и объектами уровня логики приложения (например, контроллерами) при применении рефакторинга Дублирование видимых данных.



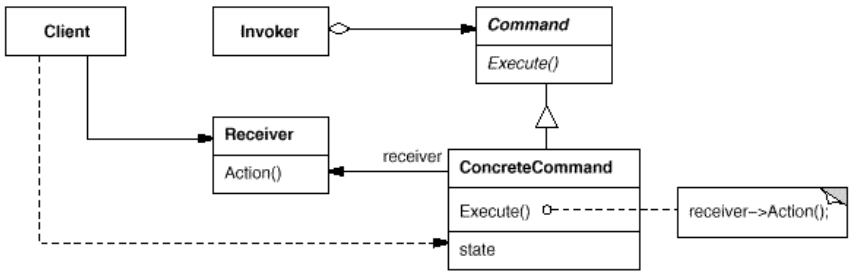
Участники : Объект, за состоянием которого наблюдают один или несколько Конкретных

наблюдателей. Объект оповещает об изменении своего состояния всех своих наблюдателей

через интерфейс Наблю-датель (то есть он не знает, какие конкретно объекты за ним наблюдают, поскольку имеет дело только со списком объектов с интерфейсом Наблюдатель и знает только об этом интерфейсе). Какой-то внешний объект (или сам наблюдатель) регистрирует конкретного наблюдателя в списке наблюдателей Объекта с помощью метода Attach. На схеме представлена модель “вытягивания” данных из конкретного Объекта самим наблюдателем. Возможна также альтернативная модель “толкания” данных, когда наблюдаемый Объект сообщает об изменившихся данных через интерфейс метода Update.
Часто используемый пример: наблюдатели - это объекты пользовательского интерфейса, а наблюдаемый объект – объект предметной области или уровня приложения (например, контроллер прецедента).

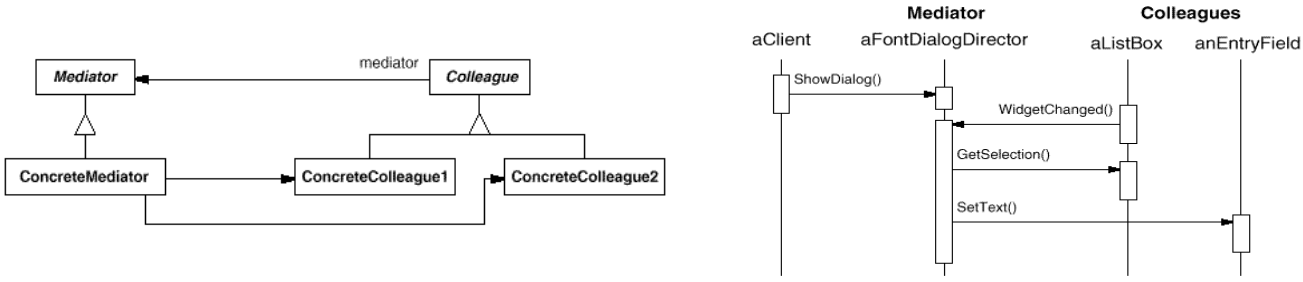


11. **Команда -** Инкапсулирует запрос в виде объекта, обеспечивая параметризацию клиентов типом запроса, установление очередности запросов, протоколирование запросов и отмену выполнения операций.

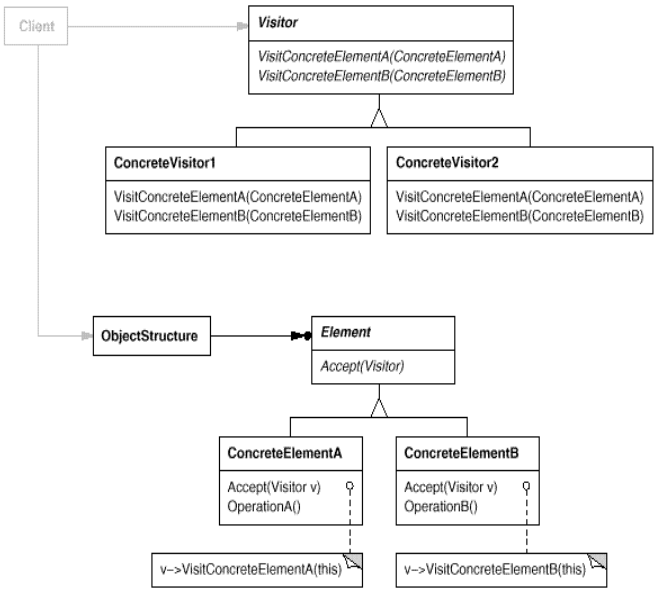


Участники: Клиент создает объект ConcreteCommand и устанавливает для него полу-чателя (Receiver). Инициатор команды (Invoker) сохраняет у себя ссылку на команду, которую он вызывает (кто-то сохраняет эту ссылку). Инициатор команды при необходимости вызывает команду через интерфейс Execute(), а конкретная команда обращается за выполнением каких-то действий к получателю. Т. о., разрывается непосредственная. связь между инициатором и получателем. Запрос становится объектом с параметрами и поведением. Их можно протоколировать, ставить в очередь, отменять, вызывать из разных мест и пр.

12. Посредник (Mediator) – Определяется объект, инкапсулирующий способ взаимодействия набора объектов. Вместо того, чтобы ссылаться друг на друга непосредственно, объекты работают через посредника. (Смотрите структуру паттерна и пример диаграммы последовательности.)



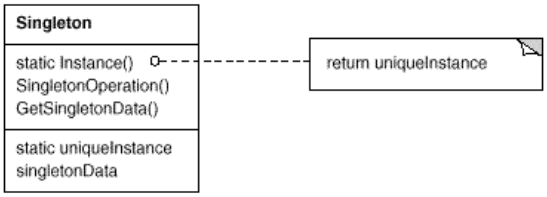
13. Посетитель (Visitor) – Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Посетитель позволяет определить новую операцию, не изменяя классы этих объектов.



Участники : Visitor – Посетитель- объявляет операцию Visit для каждого класса ConcreteElement в структуре объектов. Имя и сигнатура этого метода определяет класс, который посылает запрос Visit. Concrete Visitor – конкретный посетитель, Element – интерфейс, определяющий операцию приема посетителя Accept, Concrete Element – при посещении его вызывает операцию посетителя, соответствующую своему классу и передает ссылку на себя, чтобы посетитель мог определить его состояние.

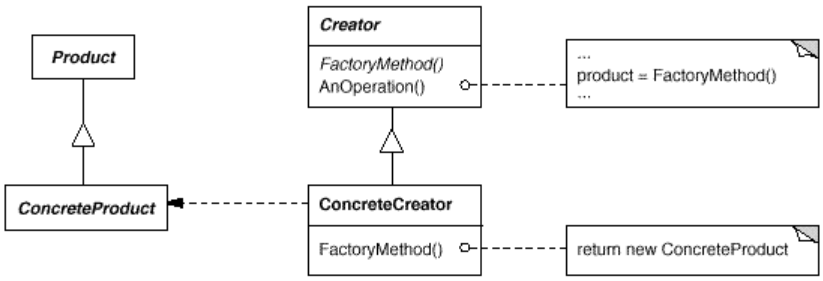
III. Паттерны создания

14. Одиночка - Для выбранного класса обеспечивает выполнение требования единственности экземпляра и предоставления к нему полного доступа.



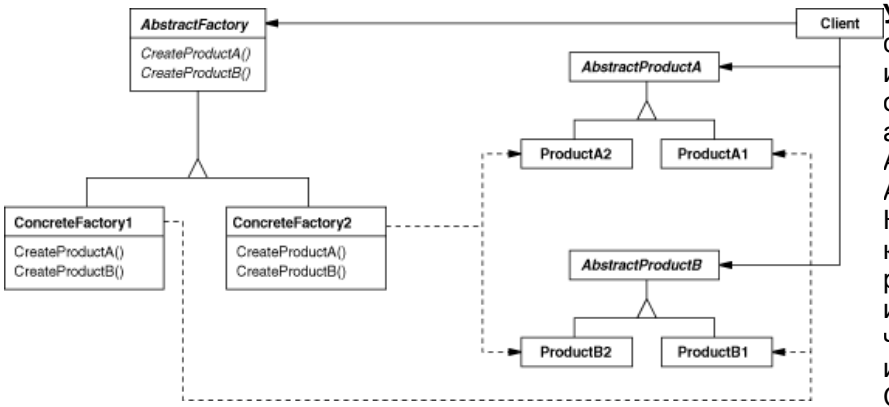
Одиночка обеспечивает существование объекта до начала его использования. На языке C++ этот шаблон реализуется с использованием статических переменных.

15. Фабричный метод – Определяет интерфейс для создания объекта в классе, но позволяет наследникам этого класса определять, объект какого конкретного класса создавать.



Участники : Интерфейс создаваемого объекта – Product, конкретный класс, реализующий интерфейс – ConcreteProduct, Создатель - класс, содержащий фабричный метод, возвращающий Product, может сам же его использовать, Конкретный создатель – наследник Создателя, реализующий фабричный метод и возвращающий конкретный продукт. Применяется часто с паттернами Композит и Итератор, в сочетании с паттерном Шаблонный Метод, развитие этого шаблона дает паттерн Абстрактная фабрика.

16. Абстрактная Фабрика - Шаблон создания экземпляров классов. Предоставляет интерфейс для создания множества связанных между собой или независимых объектов, конкретные классы которых неизвестны.



Участники : Абстрактная фабрика - определяет интерфейс для создания объектов, причем - в терминах абстрактных классов AbstractProductA, AbstractProductB и т.д. Конкретные фабрики - создают конкретных наследников, реализующих затребованные интерфейсы. Этот паттерн часто используется при использовании паттернов Стратегия, Наблюдатель и др. Можно сказать, что интерфейс Абстрактной фабрики в основном состоит только из Фабричных методов.

Тема 12 Специальные вопросы : проектирование пользовательского интерфейса и интерфейса с базой данных с использованием шаблонов.

Мы уже не раз упоминали про использование тех или иных шаблонов (проектирования, архитектурных и даже - шаблонов рефакторинга) для решения многочисленных проблем, связанных с реализацией пользовательского интерфейса и подключением к БД. Однако, эти вопросы настолько важны и нетривиальны, кроме того, с ними сталкивается практически любой разработчик, поскольку они возникают практически в 90% процентах проектов, что имеет смысл рассмотреть их отдельно.

24) **Пользовательский интерфейс (UI).**

Пользовательский интерфейс в том или ином виде присутствует у большинства (я бы даже сказал – у подавляющего большинства) приложений. Он предназначен для взаимодействия с пользователем в процессе реализации прецедентов системы. Варианты интерфейса :

33) Консольный (интерфейс командной строки). В данном случае интерфейс может сводиться к параметрам вызова программы, но может быть и интерактивным (назовем его консоль). Это достаточно бедный интерфейс, как в плане изобразительных средств, так и функционально, по сути своей – последовательный, не многозадачный. Применяется там, где это оправдано либо техническими обстоятельствами, либо спецификой приложения (терминал, сетевая ОС, утилиты и пр.)
34) Оконный интерфейс, в том числе – графический оконный, многооконный (условно обозначим его GUI). Самый богатый в плане выразительности и функциональности, как правило – управляемый событиями, многофункциональный и пр.
35) Web-интерфейс (Web). Отличается тем, что, прежде всего, предназначен для Интернет/Интранет, но не только. В данном случае интерфейс сочетает простоту создания (и изменения !) с достаточно развитыми возможностями внешней программы – броузера web-страниц. Постепенно играет все большую и большую роль с развитием Интернет и интранет-технологий.
36) Мобильный интерфейс – предназначен для мобильных устройств типа телефонов, цифровых и видеокамер, КПК и пр. Также начинает играть все большую роль в связи с увеличением кол-ва и возможностей мобильных устройств. В настоящее время представляет собой что-то среднее между названными выше.

Сам интерфейс в данном контексте можно условно разделить на 2-3 составляющие: это само внешнее отображение интерфейса (тексты сообщений, пункты меню, варианты выбора, формы ввода, картинки, гипертекстовые ссылки и пр.), код, обслуживающий интерфейс (начальная проверка вводимых данных, обработчики событий от устройств ввода, вызов диалогов, создание списков выбора, смена форм и страниц и т. д.) и еще я бы выделил собственно код, управляющий сценариями прецедентов (последовательность задаваемых вопросов / выдаваемых экранных форм, последовательность вопросов / ответов и пр.), то, что можно назвать интерактивом. Это разделение важно для дальнейшего рассмотрения.

Мы рассматриваем в общем не задачу создания интерфейса, а задачу сопряжения его с остальным проектом (назовем остальную часть просто приложением). Напомню, что согласно концепции слоев наш проект не должен быть размещен внутри кода, реализующего интерфейс, независимо от его сложности, целей и задач. Исключение составляют совсем тривиальные проекты, учебные программы и прототипы, которые создаются, в частности, для проверки каких-то вопросов использования чего-то, либо для апробации и выбора вариантов самого интерфейса. В остальных случаях у нас возникает задача сопрячь имеющийся код, иерархию классов приложения (предметную область, либо – слой технических служб и пр.) с UI. Как это сделать ?

1. Простейший вариант – разместить код создания / вызова методов классов

приложения в коде интерфейсного слоя. Заметьте, только код создания и вызова методов, а не содержание самих методов, что существенно ! Этот вариант самый плохой с точки зрения разделения слоев, но лучше уж так, чем совсем никак. В чем его недостатки ? Во-первых, в этом случае мы делаем код интерфейса зависящим от классов приложения, подчас – от всей иерархии классов. При изменении иерархии и интерфейсов отдельных классов (модулей) нам придется соответственно менять и интерфейс, по крайней мере – его часть. Косвенно имеется зависимость и у модулей приложения от модулей интерфейса в том смысле, что эти модули создаются, инициализируются и вызываются из интерфейсных, что, в принципе, не так уж важно, хотя бывает, что это приводит к необходимости изменения интерфейса модулей приложения при изменении интерфейса. Преимущество такого подхода, прежде всего – простота. Для простых проектов, а также на начальных этапах разработки можно пойти на такой вариант. Какие шаблоны здесь нужны ? Ну, в каком-то смысле Адаптер для согласования имеющихся интерфейсов, возможно Фасад.

Рис. 1

2. Более сложный вариант – использование специальных промежуточных объектов (классов, модулей) для сопряжения UI с приложением. Часто такие объекты обладают свойствами **контроллеров**. Этот шаблон не описан в явном виде у GoF, там упоминается концепция **MVC**, в которой контроллер играет важную роль. Авторы отмечают, что контроллеры реализуются совокупностью шаблонов, таких как Стратегия, Наблюдатель и др. Я бы добавил сюда шаблон Фасад, возможно – Посредник. Более подробно контроллеры описаны в книгах Лармана и Фаулера. Там рассматривается сам шаблон контроллер, а также его модификации в архитектурных шаблонах (контроллер страниц, приложения, прецедентов, слой служб). Мне ближе модель, в которой контроллер рассматривается как посредник между UI и остальным приложением, обладающий функциями контроллера прецедента или ряда прецедентов, то есть инкапсулирующий логику сценария прецедента, интерактива, насколько это возможно. Но в рассматриваемом варианте это возможно только отчасти. В данном случае предполагается, что контроллер может являться единственным объектом приложения, на который ссылается соответствующий модуль UI. UI знает о контроллере, контроллер знает, либо не знает о UI. Контроллер знает об объектах предметной области, возможно – о технических службах и т.п. Это соответствует концепции слоев.

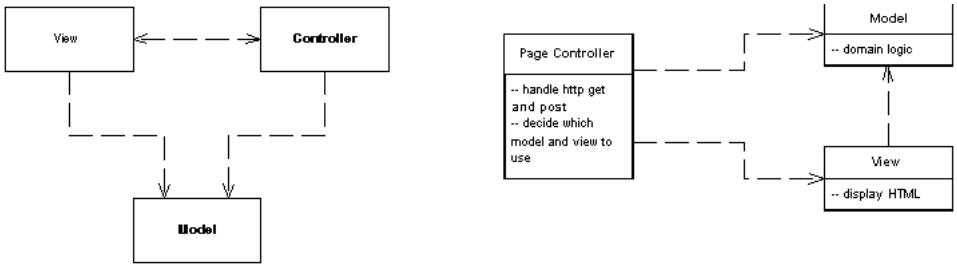


Рис. 2 MVC, Page Controller

Элемент интерфейса, допустим, форма (диалог) создает контроллер (либо – снабжается ссылкой на созданный кем-то контроллер), и взаимодействует с ним, при этом взаимодействие происходит непосредственно, но инициатором взаимодействия всегда является UI. В свою очередь, контроллер взаимодействует с объектами приложения, домена, службами и пр. Вариации контроллера : контроллер страницы (формы), контроллер приложения (прецедента), входной контроллер и пр.

Преимущество: вводится доп. уровень косвенности, делегирования сообщений, что позволяет несколько снизить зависимость UI от классов приложения, например, от классов предметной области и технических служб. Недостатки : необходимо создавать новые классы контроллеров. Кроме того, контроллеры в данном случае играют относительно пассивную роль, так как сценарий интерактива управляется по-прежнему кодом, размещенным в модуле

UI.

Здесь используются шаблоны Контроллер, возможно – Фасад, возможно использование Дублирования видимых данных (это шаблон рефакторинга !), при котором данные домена и формы дублируются для облегчения преобразования типов, проверок, отмены и пр.

3. Наконец, при последовательной реализации концепции MVC мы приходим к такому варианту взаимодействия UI и приложения, при котором контроллер приобретает активные функции и содержит по крайней мере часть логики сценария прецедента. В этом случае в нем инициируются те или иные действия в зависимости от внешних событий. Зависимость контроллера от UI в данном случае разрешается с помощью принципа **DIP** и шаблона **Наблюдатель**. Наблюдателями для контроллера становятся объекты UI, которые подписываются на события, приходящие от него. В результате контроллер зависит от интерфейса наблюдателей, но не от конкретных наблюдателей (форм, окон, меню, и пр.) Это позволяет контроллеру посылать сообщения всем, кто их может обработать. Обратная зависимость может быть непосредственной, то есть UI может знать непосредственно о своих контроллерах, а может пользоваться только интерфейсом наблюдаемого объекта (Subject).

Рис. 3

Для организации косвенной связи между объектом UI и контроллером может использоваться и шаблон Команда, особенно когда один и тот же контроллер должен вызываться из разных мест приложения. Для обеспечения возможности контроллеру передавать данные интерфейсу, не зная о его конкретной реализации, может использоваться технология дублирования данных.

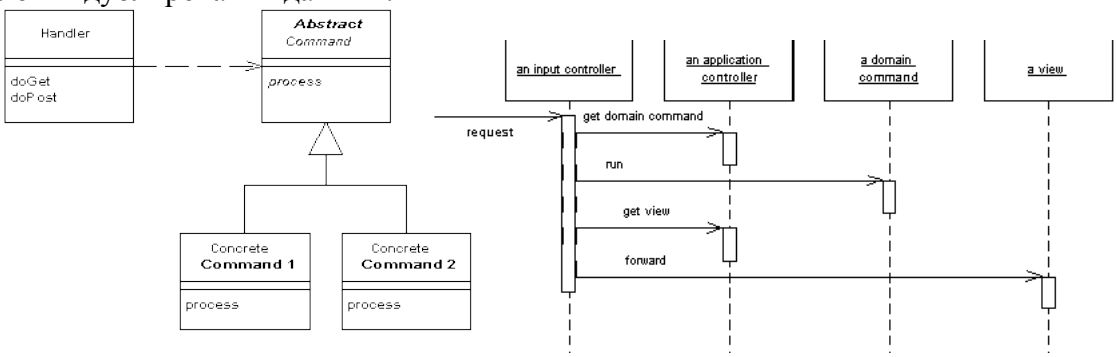


Рис. 4 **Front Controller**, **Application Controller**

Этот 3 вариант в целом является более сложным в реализации и требует больших накладных расходов, которые оправдываются в случае, когда приложение сложное и обладает сложным интерактивом, который имеет смысл переносить из уровня UI в уровень приложения и тем самым защищать его от изменений вида интерфейса, используемых технологий, платформ и т.д., то есть когда планируется повторное использование сценариев.

Преимущества : гибкость, повторное использование сценариев, тонкий клиент. В конечном итоге нужно отметить еще один плюс такого подхода : при нем легче выделить в одно место весь код для работы с конкретным интерфейсом, будь то GUI или Web.

Недостатки : сложность реализации, большое количество дублирующихся данных, снижение производительности. Последнее обстоятельство часто не играет роли, однако когда речь идет о заполнении таблицы из 1000 строк лишний уровень может оказаться чересчур медлительным.

Пример. Примером, иллюстрирующим сказанное, может служить небольшое приложение, построенное на базе задачи расчета платежей по кредиту. Вы рассмотрите его на лабораторных более подробно, в частности, те три варианта, о которых здесь идет речь.

Помимо указанных шаблонов используются также и другие, например, Компоновщик – для единообразной работы с простыми и составными объектами, в том числе и самого пользовательского интерфейса, Итератор – для сокрытия организации коллекций и составных объектов, **Маппер** (шаблон для отображения плоского представления в объектное

и наоборот), Посетитель, например, для создания отчетов и другие.

25. Взаимодействие с БД.

Приложения, работающие с БД, составляют до 80% оригинального ПО, разрабатываемого в мире. При этом варианты использования технологии БД могут быть различны :

- приложения, построенные вокруг БД (информационные системы), т.н. DDA (Data-Driven Applications);
- корпоративные системы, использующие корпоративные БД, возможно, разные, возможно, наоборот, использующие одни и те же БД по-разному, как источники данных и долговременную память для хранения данных программы между сеансами;
- системы использующие технологию БД как вспомогательную для хранения данных (в т.ч. объектов) между сеансами своей работы, в конечном счете часто возможен выбор в данном случае между СУБД или другим источником данных. В общем случае подобный выбор существует почти всегда, по крайней мере, вместо БД в целях отладки приложения может использоваться подставной источник данных, что приводит к мысли о множественности таких источников и соответственно – об использовании абстракции типа «Источник данных», о которой уже шла речь на 2 лабораторной и при рассмотрении концепции слоев.

Вообще нужно сказать, что современные СУБД являются достаточно сложными системами, которые часто подвержены изменениям, во-первых, из-за обилия используемых технологий, а во-вторых, из-за частых потребностей в изменении хранимых данных.

Действительно, управление базой данных, включает по крайней мере следующие уровни :

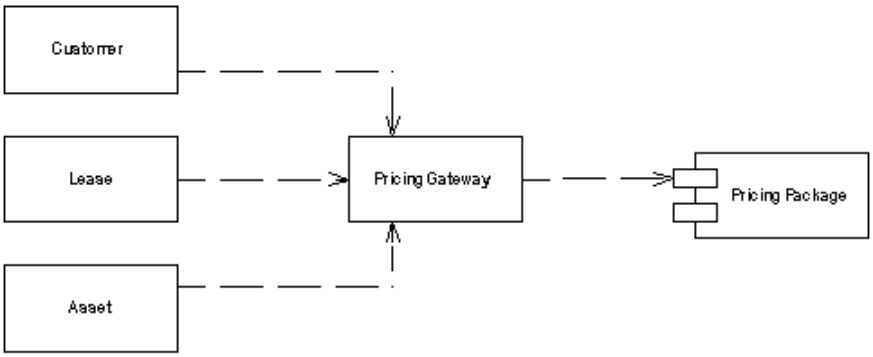
- приложение;
- библиотека доступа к данным (API);
- сервер БД или какой-либо SQL- движок.

В отмеченных выше условиях постоянных изменений структуры баз, а также используемых технологий (последние могут меняться из соображений новых версий, прекращения поддержки, изменения условий лицензирования, выявленных проблем, переноса данных на другую платформу, публикации данных в Интернет и пр.) так же, как и в случае с интерфейсом, нельзя допускать размещение кода для доступа к конкретным таблицам конкретных БД с помощью конкретных API в коде предметной области и приложения в целом, нежелательно также размещение такого кода в UI, чему способствует использование визуальных сред разработки **(RAD).**

Что здесь можно предложить ? По-прежнему, введение дополнительной косвенности. Здесь существует по крайней мере 2 уровня косвенности : первый уровень – **широкое использование SQL**. SQL является стандартом для различных СУБД, поэтому замена обращений к таблицам через интерфейс библиотеки доступа к данным на вызов SQL запросов может значительно улучшить ситуацию, по крайней мере для начала. Дополнительный механизм в данном случае – **использование хранимых процедур**.

С другой стороны, это не решает всех проблем, так как ссылки на таблицы (или – на сохраненные процедуры) остаются в коде приложения и делают его зависимым от БД. Поэтому 2 уровень косвенности – **это использование отдельного слоя источника данных.** (См. лабораторную № 2). В книге Фаулера он упоминается как базовое типовое решение **Шлюз** (шлюз определяется как **фасад** для доступа к внешним системам или ресурсам). Он позволяет, во-первых, собрать весь код доступа к БД в одном месте, во-вторых, освободить код приложения от связи с БД.

Само по себе разрешение зависимостей выполняется, во-первых, за счет шаблонов типа **Фасад**, либо, в более сложных случаях – типа **Заместитель (Proxy)**, а во вторых – использованием принципа инверсии зависимостей DIP, как мы рассматривали ранее.



Для обозначения этого приема в терминах шаблонов используют наименование шаблона **Абстрактный сервер** (Мартин).

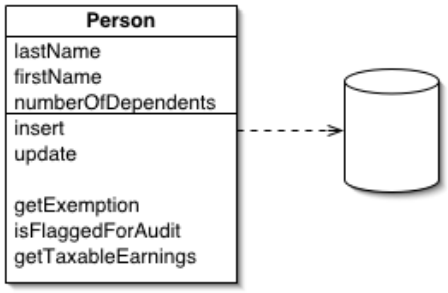
Остается еще один немаловажный вопрос: а как реализовать этот фасад (шлюз) либо прокси, либо другой шаблон, с учетом дополнительной проблемы: преобразования объектных данных в реляционные (плоские) и обратно ? (Понятно, что специфику сервера СУБД или API можно спрятать за фасадом, но как быть с полями, связями и т.п. ?!)

Существует большое количество архитектурных шаблонов для решения данных проблем, которые относятся к шаблонам источников данных и объектно-реляционного отображения. Они, в частности, описаны у Фаулера в книге “Архитектуры корпоративных программных приложений”.

Вот шаблоны источников данных :

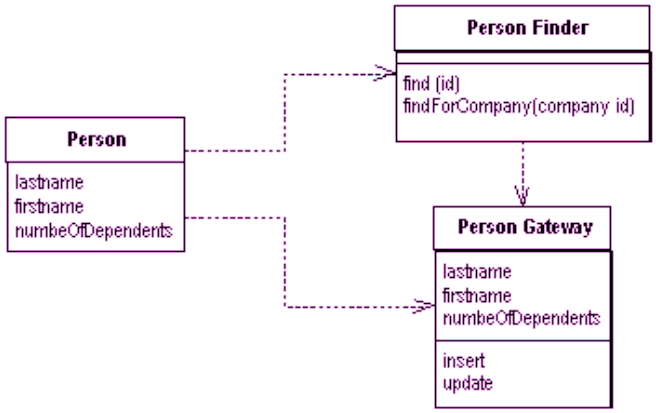
7. Активная запись
8. Шлюз строки.
9. Шлюз таблицы.
10. Преобразователь данных.

Активная запись :



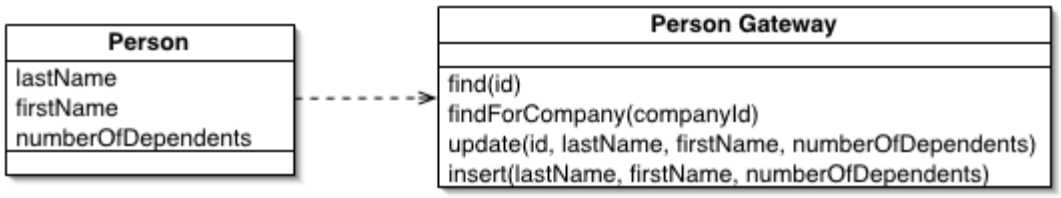
Самый простой вариант, при котором сама класс предметной области (домена) снабжается методами для сохранения и загрузки себя из БД. Поддерживается концепция слоев, но слой доступа к данным смешан со слоем домена, единственная абстракция (то есть уровень косвенности) – это SQL, и, возможно – механизм хранимых процедур.

Шлюз строки данных :



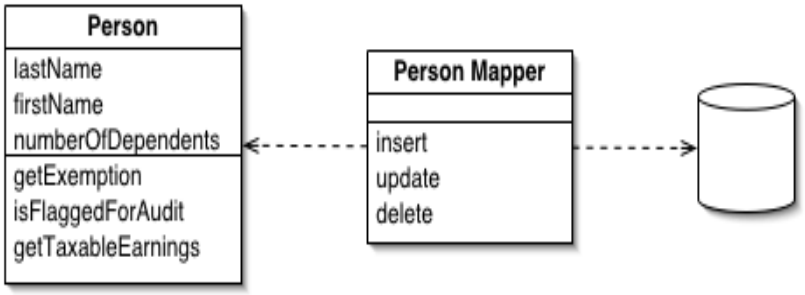
В этом решении создается **отдельный** класс, который инкапсулирует методы сохранения данных об одном объекте и загрузки этих данных. При этом объект соответствует одной строке таблицы.

Шлюз таблицы :



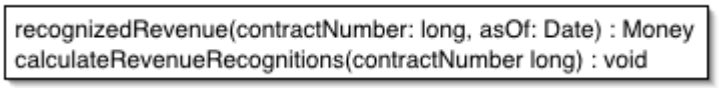
В данном случае инкапсулируются методы для работы со всей таблицей, то есть с набором объектов.

Преобразователь данных (Mapper):

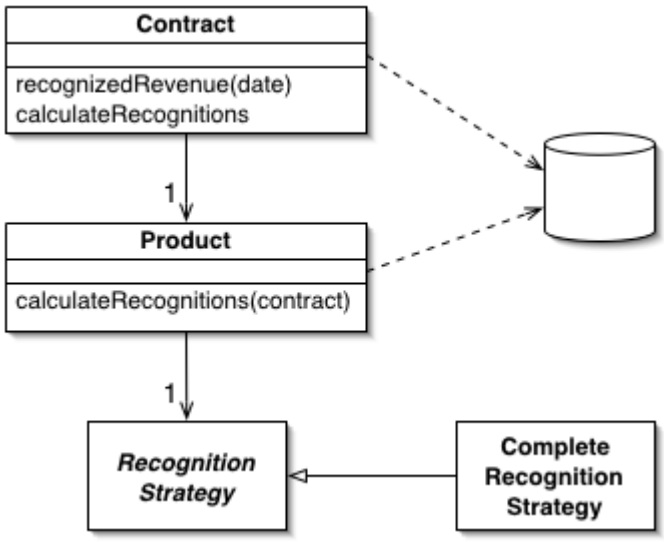


Это самый сложный шаблон источников данных, поскольку он должен знать и о БД, и о структуре домена, и постоянно корректироваться при изменении того или другого.
На самом деле задача преобразования большого кол-ва полей в большое кол-во атрибутов (да еще и с учетом связей и иерархии) является нетривиальной и однозначного рецепта для ее решения нет. Преобразователь данных является самым гибким, но и самым тяжелым для реализации. Немного облегчить ситуацию может использование **преобразования на основе метаданных (**то есть данных о данных**)**, но при этом придется повозиться на первом этапе при создании такого преобразователя.
Часто на первый план при работе с БД выходит вопрос производительности, который в данном случае является не лишним. С этим связан и другой вопрос, который уже поднимался в нашем курсе – вопрос о целесообразности наличия бизнес-слоя (модели предметной области) в информационной системе (СУБД) и о месте размещения бизнес-алгоритмов и бизнес-правил: в коде хранимых процедур, либо – в коде объектов домена.
Ответ на этот вопрос также неоднозначен. Действительно, он связан с шаблонами организации самой предметной области, которые также рассматриваются у Фаулера :

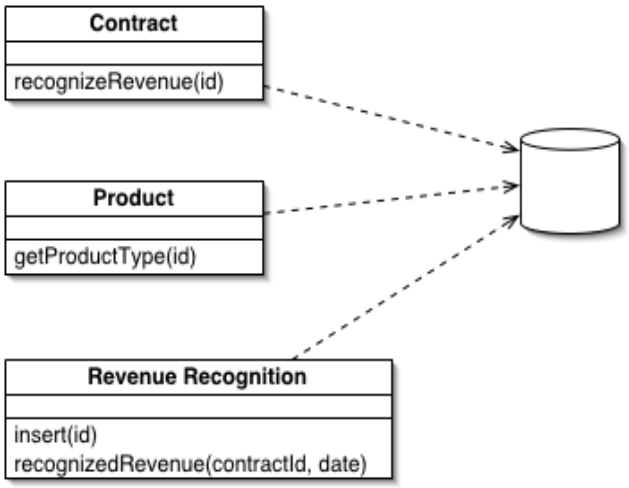
4. **Сценарий транзакции (нет предметной области)** :



5. **Табличный модуль (классы ПО строятся вокруг таблиц БД)** :

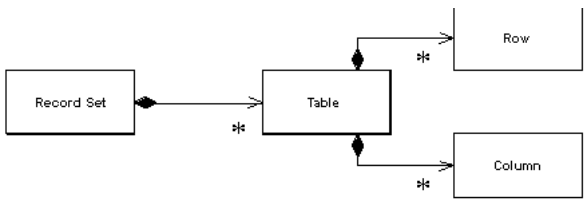


6. **Модель предметной области (иерархия классов ПО сама по себе, а БД – сама по себе)**:



Соответственно первый вариант может использоваться с разными источниками данных, это непринципиально. Второй вариант хорошо согласуется с шлюзом таблицы, а для третьего в сложных случаях подойдет только преобразователь данных, в более простых – другие шлюзы и шаблоны.

Еще один типичный вопрос, возникающий при использовании шлюзов, фасадов, слоев и прочих подобных шаблонов в контексте БД : как передавать между компонентами системы табличные данные ? Для этого можно использовать типовой шаблон **Набор записей (RecordSet) :**



Важность этого шаблона состоит также в том, что именно он позволяет реально отделить БД от UI, и при этом часто позволяет сохранить производительность на

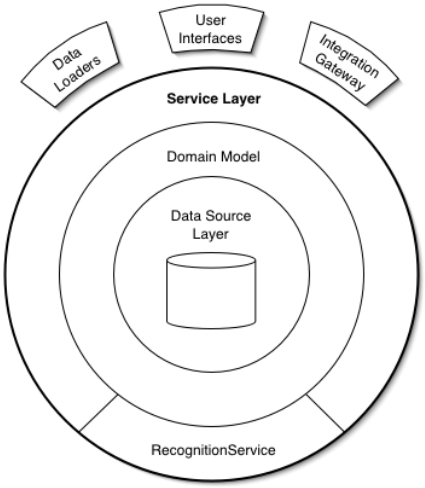
приемлемом уровне ! (Пример 1 : ADO + MFC + grids. Для ускорения вывода больших таблиц можно приделать шаблон Адаптер (классов!) к стандартному классу ADO::Recordset для обеспечения своего интерфейса RecordSet, наряду с которым будет использоваться его собственный интерфейс, и он сможет пониматься как доменом, так и гридом !).
(Пример 2: **.NET.** В платформе .NET работа с БД реализуется с помощью библиотеки ADO.NET, в которой шаблон RecordSet реализован в явном виде как **DataSet** – представление таблиц в памяти. Это принципиально отсоединенный от БД набор таблиц. Остается по-прежнему зависимость от платформы, которую можно в некоторой степени устранить с помощью адаптеров объектов.)

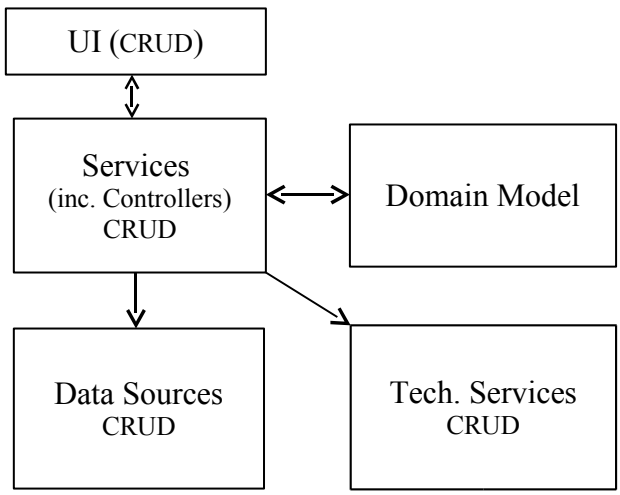
Раз мы заговорили о .NET и способах передачи данных, в т.ч. Табличных и иерархических, то нельзя не упомянуть про **XML**. XML позволяет сериализовывать в виде специально отформатированного (размеченного) текста объекты, наборы объектов, таблицы и пр. в стандартном формате ! Платформа .NET очень широко использует XML и преобразования из XML в DataSet и обратно. Конечно, при этом несколько снижается производительность, но зато возрастает гибкость и интероперабельность. XML является основным форматом для передачи данных между независимыми слоями. Также XML является форматом, в котором возвращают данные XML Web – сервисы, являющиеся центральным компонентом технологии распределения приложений Microsoft в наст. время.

Рис.

Тем не менее, при рассмотрении этих вопросов часто возникает следующее опасение (я о нем уже говорил) : роль объектов домена и других часто сводится к перекачке данных, на это тратится время и силы разработчиков, а потом – компьютера, и не всегда понятно, зачем это нужно !

И тут необходимо вспомнить еще об одном шаблоне : **Слой служб**. Он также описан у Фаулера. В нем говорится о том, что если задача форм UI сводится к т.н. операциям CRUD (CREATE – READ - UPDATE - DELETE) над таблицами БД, то можно поручить эту задачу слою служб, который выполняет CRUD без домена и без сложных преобразований из объектов в реляционное представление и обратно. Реализуется слой служб техническими объектами + контроллерами.





Слои служб

В общем случае можно говорить и о том, что предметная область в проекте, связанном с данными может и отсутствовать! Но почти всегда можно найти применение и ей, и, особенно, слою служб и контроллеров. Иногда такая потребность возникает и на более поздних стадиях проекта, поэтому надо быть готовыми к тому, чтобы решить эти новые проблемы с помощью описанных шаблонов !

Типичные причины возникновения модели предметной области в проекте :

- сложные алгоритмы проверки данных до ввода в БД;
- необходимость обработки данных НЕ на сервере БД (в памяти, либо – только на

клиентской машине), в частности – накопление данных на клиенте при аварии сервера;

- расчеты, которые трудно реализовывать средствами SQL;
- перспективы частого изменения интерфейса и используемых технологий доступа к данным (ядро нескольких проектов, перенос проекта в Web, проект для массового потребителя и пр.), одним словом – необходимость повторного использования разрабатываемого кода в разных контекстах и пр.

Если же вас беспокоит производительность, то в ряде случаев на помощь могут придти технологии распределенной обработки ! Они опираются на модель слоев, причем предполагают еще большее расслоение, причем теперь слои не просто располагаются в разных классах, но и в разных модулях, а модули – на разных вычислительных узлах сети.

Тема 13 Упаковка программных проектов. Метрики упаковки (см. Лаб. Работу № 7).

Тема 14

Построение распределенных приложений (технологии, платформы, протоколы, интерфейсы)

В ходе нашего курса мы уже упоминали некоторые особые виды приложений, которые можно условно отнести к информационным, а также к корпоративным. В основном примеры в данном курсе, частично и ваши семестровые относятся к данным видам систем. В последнее время в связи с развитием сетевых технологий, Интернет, а также – мобильной техники все большое зхначение приобретают т.н. **распределенные приложения (Distributed Applications).**

Под **распределенными** будем понимать приложения (РП), которые размещаются на разных вычислительных узлах (минимум на 2). Для реализации таких приложений необходимо решать огромное количество вопросов. Так или иначе они касаются аспектов:

1. Логической декомпозиции проекта (модули, процедуры, классы, пакеты).
2. Физической декомпозиции проекта (программные единицы, модули, страницы, СП, библиотеки, исполняемые и интерпретируемые файлы).
3. Распараллеливания вычислений и алгоритмов работы (многопоточность, многозадачность, параллельные вычисления и пр.)
4. Физической и протокольной организации коммуникации компонентов системы, размещенных на разных узлах (протоколы, интерфейсы, способы взаимодействия).
5. Межплатформенного взаимодействия (работа узлов под управлением разных ОС и пр.)
6. Защиты информациии и аутентификации пользователей.
7. Защиты данных, обеспечения целостности данных и организации совместного доступа к данным.

И этот перечень нельзя назвать полным. Так или иначе к распределенным приложениям как правило относятся большинство информационных систем, все корпоративные системы, и даже относительно простые интернет-системы и многие мобильные проекты.

В связи с этим можно рассматривать почти ВСЕ проекты (за исключением, пожалуй, утилит и др. специализированного ПО) как распределенные или потенциально распределенные системы. То есть распределенное приложение – это просто абстракция приложения вообще, а локальная программная система – это ее вырожденный случай.

Мы, конечно, не можем рассмотреть даже большую часть упомянутых аспектов, но хотя бы постораемся указать, гда и как искать ответы на вопросы.

Ну, во-первых, отметим, что аспекты 1 – 2 рассматривались нами в ходе всего курса. Рассмотренные нами концеции архитектурных шаблонов, организации многуровневых приложений, декомпозиции и пакетирования систем носят общий характер, как и концепции ООП, в том числе они непосредственно укладываются в концепции распределенных приложений.

К сожалению, 3 аспект полностью не рассмотрен нами, но вы его частично рассматривали в курсах Операционные системы, возможно – в СПО и ПО компьютерных сетей. Хотя эти задачи, конечно, требуют отдельного рассмотрения. Если говорить об ответсвенности за реализацию этого аспекта, то она распределяется между разработчиками и системным ПО.

Аспект коммуникации компонент системы также частично рассматривается в других курсах : сети, ПО сетей, Web-программирование. Краешек этого аспекта показан и в нашем курсе. (пример в 7 лабе). Сейчас мы его чуть-чуть затронем. Здесь также присутствует двойная ответственность. Важно отметить, что использование рассмотренных нами концепций построения программных систем облегчает реализацию коммуникации, что наглядно иллюстрирует последняя лабораторная работа.

Межплатформенное взаимодействие - также очень важный аспект. Мы его, к

сожалению, не успели рассмотреть, но чуть-чуть осветим. И здесь ответсвенность распределена между системным ПО, используемым готовым решениям для разработчиков и между самим разработчиками. Опять-таки, грамотное следование концепциям декомпозиции проекта облегчает решение и этих задач !

Что касается вопросов защиты информации и данных, обеспечения совместного доступа к ним, то тут вы, я думаю, согласитесь, что большая часть ответственности перекладывается на готовые системы, такие как сервера БД, серверные ОС и пр. (Курсы ОС, БД, сети, сетевое ПО). Тем не менее, ответственность разработчика сохраняется и здесь.

Как вы видите, действительно, первые из указанных аспектов доминируют (за исключением 3 !) над остальными и частично определяют пути, варианты решения этих вопросов. К сожалению, часто разработчики (особенно неопытные и не знакомые с принципами ООП и декомпозиции систем) попадают в плен готовых решений, которые претендуют на комплексное решение вопросов построения РП.

На самом деле, как вы можете убедиться на простом приведенном примере, иногда для построения распределенного приложения не требуется каких-то сверусилий, при условии, что приложение правильно спроектировано и упаковано ! Достаточно использовать определенный протокол и среду передачи, адаптировать (шаблон Адаптер-Прокси) свой прикладной протокол к нему – и все ! В примере к л/р я использовал библиотеку сокетов и протокол TCP/IP. Все остальное – это неизмененное наше приложение !

Конечно, в этом случае приходится изобретать велосипед и при раширениии приложения все усложняется. Для облегчения жизни разработчиков предлагается масса КОНКРЕТНЫХ технологий. Безусловно, их нужно изучать !!! Вы не сможете профессионально решить вопросы 3-7 аспектов на уровне готовых платформ ! Но вы дожны иметь представление как это делается. Именно поэтому мы основное внимание уделяли аспектам 1-2.

Что же касается конкретных технологий, то, прежде всего, необходимо рассмотреть платформы и готовые технологии распределения, на которые они опираются :

1. Платформы для разработки распределенных корпоративных приложений.

Их существует несколько, но основных, пожалуй, две : Microsoft.NET, и J2EE. Эти две платформы претендуют на лидерство, и пока трудно сказать, какая преуспевает. Они базируются, соответственно на платформах .NET Framework и Java 2.

2. **Уровни технологий распределения:**

1. Языки разметки текста (SOAP, XML).

К этому же уровню следует отнести популярные форматы представления данных в Web, прежде всего – XML. XML позволяет передавать структурированные массивы данных в виде текста с разметкой между различными стандартными системами и даже между платформами. Конечно, мы должны были бы рассмотреть XML, но увы...

2. Протоколы прикладного уровня (например, HTTP).

3. Технологии распределения объектов, такие как : DCOM, CORBA.

4. Технологии удаленного вызова методов : (например, RMI).

5. Технологии удаленного вызова процедур : RPC.

6. Протоколы транспортного уровня (TCP-UDP) с использованием интерфейса сокетов.

Немного о технологиях построения распределенных приложений в двух основных

платформах :

1. J2EE – Java 2 Enterprise Edition

Эта платформа, пожалуй, является лидером по количеству внедренных крупных корпоративных распределенных проектов на ее базе. Базируется платформа на языке Java 2, который является, пожалуй, единственным реально мультиплатформенным языком благодаря наличию виртуальной машины JVM и ее реализации для большинства известных платформ, ОС, для мобильных систем (в версии J2ME). Приложение на Java 2 преобразуется в байт-код, исполняемый JVM. В результате приложение без повторной сборки может переноситься на другую системную платформу. С точки зрения поддержки распределенных приложений платформа J2EE предлагает технологию POJO (Portable Old Java Objects), более классическую (RMI), в сочетании со средством доступа к данным JDBC, а а также более новую технологию EJB (Enterprise Java Beans), аналог COM/COM+/DCOM. Интерфейс пользователя реализуется с помощью библиотек типа Java Swing и др. Поддержка серверных страниц – JSP/Servlets. Естественно, платформа не стоит на месте и в нее интегрируются все новые веянья, типа Web-сервисов (JAX-WS) и пр. То есть, платформа предоставляет средства интеграции на самом разном уровне. Средства разработки – как очень дорогие коммерческие (типа Sun Enterprise Studio, Jbuilder, WebSphere), так и бесплатные (Eclipse). Основные локомотивы – Sun, IBM, Borland. Поддерживаются основные ОС (Linux, Windows).

2. Microsoft .NET

Эта платформа появилась намного позже J2EE, но зато представлена абсолютным лидером IT индустрии – компанией Microsoft.

Основные концепции .NET :

- Новая среда выполнения программ **.NET Framework (теперь уже 2.0) / CLR**. Среда использует свой байт-код (**MSIL**), в который транслируются приложения, затем они компилируются в исполняемый код с помощью **JIT**-компилятора и исполняются средой CLR. **CLR** - Это фактически аналог JVM – мультиязыковый рефлективный движок для запуска приложений. Кроме того, используется большой набор библиотек .NET Framework, доступный из различных языков .NET (VC++, CSharp, VB, VJ). CLR обеспечивает управление этим управляемым кодом (managed code), то есть – обращается к библиотекам .NET, поддерживает работу с общими типами, обрабатывает ошибки, даже вроде кэширует скомпилированные модули) и пр. Сейчас поддержка CLR существует более чем для 40 языков.
- Замена традиционного для Windows механизма взаимодействия программных компонентов COM новой технологией, которая сейчас называется .NET. Впрочем, некоторые эксперты считают, что в данном случае мы имеем дело скорее с развитием существующей ныне COM+, которое сопровождается сменой названий.
- Межпроцессовое взаимодействие и взаимодействие различных пространств приложений (application domains), в том числе и удаленное, реализуется с помощью технологии **.NET Remoting**.
- Новая технология реализации серверных страниц для Интернет / Интранет приложений, работающих с сервером **IIS** – **ASP.NET.** Развитие технологии ASP в лучшую сторону (Например, поддержка отделенных от представления классов страниц – Code Behind classes). В том числе – реализация **ASP.NET Web сервисов**.
- Широкое применение **XML** вообще и **XML Web Services** в частности (Как на уровне **ASP.NET,** так и **.NET Remoting)**. Но тут нужно подчеркнуть, что **XML Web Services** - это открытые протоколы, и Microsoft - лишь один из разработчиков и пользователей этой технологии.

- Усовершенствованная библиотека доступа к данным - **ADO.NET.** Во-первых, это реализация объекта “набор данных” (**DataSet**), представляющего набор связанных таблиц в памяти, не связанного при этом с БД. Во-вторых, усовершенствованная иерархия классов. В-третьих, хорошая поддержка **XML** и преобразование XML-DataSet и обратно. Все это является частью библиотеки классов .NET Framework.
- Усовершенствованные библиотеки создания пользовательского интерфейса с унифицированными компонентами для GUI интерфейса **(Windows Forms)** и ASP.NET – **Web Forms**.
- Набор инструментальных средств разработки **Visual Studio .NET**, который позволяет создавать приложения класса .NET. Это ключевое для всей концепции программное средство, и именно поэтому точка отсчета эпохи .NET однозначно привязана к официальному выпуску VS.NET (февраль 2002 г.). Новая версия .NET Framework 2.0 соответственно привязана к выпуску Visual Studio 2005 (конец 2005 года). Включает 4 базовых языка, использующих одну и ту же библиотеку классов : VC++, VB, C#, (VJ). Следует заметить, что большинство (или все) технологий .NET доступны через бесплатные Command-Line средства (компиляторы, линковщики, редакторы ресурсов, другие утилиты и пр.)
- Реализация технологий .NET в виде конкретных продуктов, составляющих платформу Microsoft : операционные системы, офисные пакеты и, конечно же, семейство серверов (SQL сервер, IIS, BizTalk, SharePoint и пр.).
- Отдельно нужно сказать об использовании технологий .NET применительно к широкому кругу мобильных систем. Быстро растущая роль мобильных устройств - это одна из ключевых идей концепции .NET.

Основные технологии построения распределенных приложений в .NET и их связь с общей картиной :

- .NET Remoting;
- ASP.NET Web-сервисы;
- ASP.NET страницы и поддерживающие их классы;
- Поддержка XML DOM и парсеры XML;
- ADO.NET;
- Сетевые возможности : сокеты, протокол TCP.
- Поддержка COM/COM+/DCOM (для существующих решений);
- Защита информации, аутентификация и т.д. - в разных библиотеках;

Подробнее :

ASP.NET Веб-сервисы

Веб-сервис (или веб-служба) - это (неформально) приложение, обрабатывающее запросы по протоколу HTTP и возвращающее ответ в виде документа XML.

В основе Web-служб лежат несколько простых принципов. Возможные для вызова команды описываются на языке WSDL (Web-Service Descriptive Language); непосредственная активизация команд происходит в виде посылки SOAP-сообщений (Simple Object Access Protocol – протокол обмена объектами, использующий XML) по адресу, где располагается Web-служба (используется стандартный протокол HTTP); для поиска Web-служб существуют глобальные или локальные (внутренние) каталоги, поддерживающие стандартные службы обнаружения UDDI. Не вдаваясь в технические подробности, можно отметить, что все современные средства разработки ведущих производителей поддерживают создание Web-служб, а программные платформы (будь то серверные операционные системы или серверы приложений) обеспечивают выполнение Web-служб.

В Web-службах везде используется язык XML. Он служит, в частности, для описания сообщений, которыми могут обмениваться Web-службы и их потребители. SOAP-сообщение — это XML-документ, состоящий из трех базовых элементов: <Envelope>, <Header> и

<Body>. Язык WSDL базируется на языке XML и позволяет создавать XML-документы, описывающие методы Web-служб, параметры методов, способы их вызова и т.п. Для того чтобы воспользоваться специализированными Web-службами в рамках механизмов обнаружения UDDI, следует составить SOAP-сообщения и интерпретировать возвращаемые XML-документы.

Жизненный цикл Web-службы можно условно разделить на три фазы: первая — программирование и публикация, вторая — поиск в каталоге, третья — потребление из клиентского приложения (рис. 1).

Рис. 1. Жизненный цикл Web-службы

Следует отметить, что, возможно, самой важной особенностью Web-служб является их независимость от платформы. Это означает, что Web-служба и ее потребитель могут быть реализованы практически на любой программной платформе, причем как сама служба, так и ее потребители могут быть реализованы на разных платформах — минимальным требованием к потребителям является поддержка протокола HTTP и возможность программной обработки XML-документов.

По мере возрастания интереса к Web-службам появлялось все больше реализаций Web-служб для решения корпоративных задач; прежде всего Web-службы применялись для интеграции приложений, для обеспечения создания функциональных модулей в гетерогенных средах, для реализации уровней абстракции бизнес-логики от клиентских приложений и т.п.

Применение Web-служб в качестве основной технологии для создания корпоративных приложений привело к тому, что базовые протоколы и стандарты, удовлетворявшие требованиям разработчиков публичных Web-служб, оказались не слишком пригодными для решения задач, стоящих перед корпоративными программистами. В результате появилась масса дополнений и расширений базовых протоколов, ориентированных на растущие потребности к повсеместному использованию Web-служб.

Для создания Веб-служб в .NET можно использовать как технологию ASP.NET, так и .NET Remoting. В первом случае неизбежно нужно использовать IIS, но этот вариант является более универсальным и платформенно-независимым. Рассмотрим его.

Для создания веб-сервиса нужно просто создать класс, наследник базового класса WebService .NET, с помощью специальных маркеров в коде выделить публикуемые методы и скомпилировать его в виде dll-сборки (assembly). Кроме того, нужно создать ASMX страницу, которая будет ссылаться на этот класс. После этого нужно опубликовать сервис на сервере IIS, при этом он должен быть связан с запущенным хост-приложением ASPNET_WP.

Все остальные действия выполняются тандемом IIS+ASPNET автоматически (правда, я не упомянул про настройки доступа и т.п., которые выполняются с помощью файла .web.CONFIG).

Доступ клиентов к веб-сервису выполняется посредством прокси-класса, который можно создать самому, сгенерировать с помощью утилит DISCO.EXE и WSDL.EXE, либо он

создастся автоматически при добавлении Веб-ссылки в проект VS.NET. После создания прокси-класса и добавления ссылки на него в проект клиента (либо в ASPX страницу) им можно пользоваться как локальным классом, обращая внимание, что замещаемый им класс на сервере является stateless, то есть обращения к нему выполняются в пакетном режиме, он не поддерживает свое состояние между обращениями (создается заново при каждом обращении), что характерно для Веб-приложений (этим можно управлять, но не в широких пределах). Веб-сервисы могут возвращать как простые типы данных, так и коллекции и другие сложные типы .NET, включая DataSet (!).

.NET Remoting

Технология .NET Remoting предоставляет довольно широкий круг возможностей по разработке распределенных приложений для платформы .NET. В отличие от ASP.NET Веб-сервисов оба участвующих во взаимодействии приложения (и клиент, и сервер) должны быть реализованы в рамках платформы .NET. Кроме того, если веб-сервисы обязательно публикуются на веб-сервере (и по всей видимости, пока только на IIS, работающем совместно с ASP.NET), то доступ к Remoted-объекту может реализовываться вашим собственным серверным хост-приложением, реализующим определенный интерфейс. Это приложение может быть графическим, консольным, сервисом NT. Сами распределенные объекты создаются в виде сборок (assemblies), помещаемых (например ?) в динамические библиотеки *.dll. Хостом может быть стандартноле приложение, например, IIS или сервер приложений.

Взаимодействие между клиентом и сервером осуществляется с помощью различных транспортов (каналов), например, TCP или протокол HTTP, причем канал можно настраивать без перекомпиляции приложений ! Кроме того, для передачи выполняется сериализация и маршалинг с помощью классов-форматеров (своих собственных, либо – стандартных: SOAP / Binary). При передаче по протоколу HTTP в отличие от ASP.NET Web-сервисов не на 100% поддерживается какой-то внешний стандарт, понятный другим платформам, поэтому даже в этом случае желательно, чтобы клиент также был реализован на той же платформе .NET. При использовании бинарного канала явным преимуществом подобной технологии перед веб-сервисами является **потенциально** более высокая производительность (с оговорками). По умолчанию удаленный объект взаимодействует с клиентом, сохраняя свое состояние, в отличие от веб-сервиса, который по умолчанию активируется заново при каждом обращении и работает в Stateless режиме (хотя ограниченную поддержку состояния можно включить). Итак, преимуществами технологии .NET Remoting являются :

- потенциально (!) более высокая производительность;
- отсутствие необходимости во внешнем сервере (помимо своего собственного);
- поддержка состояния удаленного объекта по умолчанию;
- б*о*льшая гибкость, но и б*о*льшая сложность;
- является универсальной технлоогией межпроцессовых вызовов;
- более объектно-ориентированная структура;
- и др.

С другой стороны, главным преимуществом технологии ASP.NET веб-сервисов является межплатформенная совместимость. Веб-сервисы ASP.NET показывают более высокую производительность по сравнению с .NET Remoting на базе http канала. Таким образом, и та, и другая технология могут найти и находят свое применение.

Для создания удаленного класса опять-таки нужно унаследовать его от класса MarshalByRefObject и все. Желательно, чтобы класс возвращал простые типы данных, либо .NET типы. ...

Кроме того, необходимо создать хост и конфигурационный файл для него. На стороне клиента нужно импортировать метаданные удаленного класса, включить механизм

удаленного вызова и также настроить конфигурационный файл. И все !

Детали, конечно, более сложны ...

Web-сервисы работают на уровне SOAP / XML, .NET Remoting – там же, но захватывает уровни 2-3.

Тема 15. Другие процессы : RUP (покороче, можно только определение + схему):

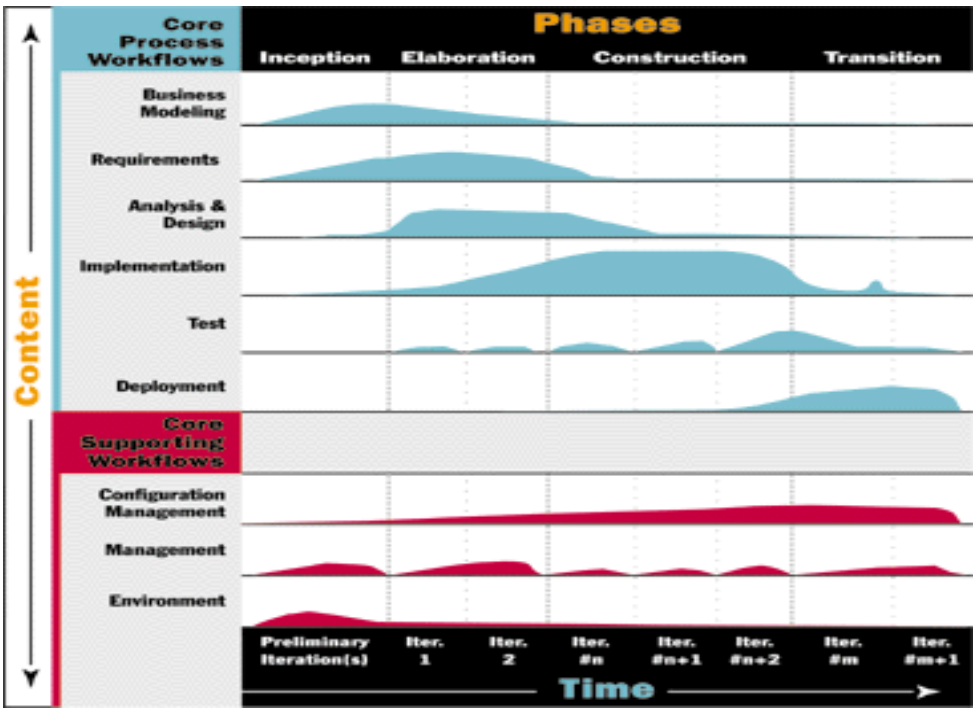
Rational Unified Process – это методология создания программного обеспечения, оформленная в виде размещаемой на Web базы знаний, которая снабжена поисковой системой.

Продукт Rational Unified Process (RUP) разработан и поддерживается Rational Software (теперь часть IBM). Он регулярно обновляется с целью учета передового опыта и улучшается за счет проверенных на практике результатов.
RUP обеспечивает строгий подход к распределению задач и ответственности внутри организации-разработчика. Его предназначение заключается в том, чтобы гарантировать создание точно в срок и в рамках установленного бюджета качественного ПО, отвечающего нуждам конечных пользователей. RUP способствует повышению производительности коллективной разработки и предоставляет лучшее из накопленного опыта по созданию ПО, посредством руководств, шаблонов и наставлений по пользованию инструментальными средствами для всех критически важных работ, в течение жизненного цикла создания и сопровождения ПО. Предоставляя каждому члену группы доступ к той же самой базе знаний, вне зависимости от того, разрабатывает ли он требования, проектирует, выполняет тестирование или управляет проектом - RUP гарантирует, что все члены группы используют общий язык моделирования, процесс, имеют согласованное видение того, как создавать ПО. В качестве языка моделирования в общей базе знаний используется Unified Modeling Language (UML), являющийся международным стандартом.
Особенностью RUP является то, что в результате работы над проектом создаются и совершенствуются модели. Вместо создания громадного количества бумажных документов, RUP опирается на разработку и развитие семантически обогащенных моделей, всесторонне представляющих разрабатываемую систему. RUP – это руководство по тому, как эффективно использовать UML. Стандартный язык моделирования, используемый всеми членами группы, делает понятными для всех описания требований, проектирование и архитектуру системы.
RUP поддерживается инструментальными средствами, которые автоматизируют большие разделы процесса. Они используются для создания и совершенствования различных промежуточных продуктов на различных этапах процесса создания ПО, например, при визуальном моделировании, программировании, тестировании и т.д.
RUP – это конфигурируемый процесс, поскольку, вполне понятно, что невозможно создать единого руководства на все случаи разработки ПО. RUP пригоден как для маленьких групп разработчиков, так и для больших организаций, занимающихся созданием ПО. В основе RUP лежит простая и понятная архитектура процесса, которая обеспечивает общность для целого семейства процессов. Более того, RUP может конфигурироваться для учета различных ситуаций. В его состав входит Development Kit, который обеспечивает поддержку процесса конфигурирования под нужды конкретных организаций.
RUP описывает, как эффективно применять коммерчески обоснованные и практически опробованные подходы к разработке ПО для коллективов разработчиков, где каждый из членов получает преимущества от использования передового опыта в:

- итерационной разработке ПО,
- управлении требованиями,
- использовании компонентной архитектуры,
- визуальном моделировании,
- тестировании качества ПО,
- контроле за изменениями в ПО.

RUP организует работу над проектом в терминах последовательности действий (workflows), продуктов деятельности, исполнителей и других статических аспектов процесса с одной стороны, и в терминах циклов, фаз, итераций и временных отметок завершения определенных этапов в создании ПО (milestones), т.е. в терминах динамических аспектов

процесса, с другой. Если попытаться представить процесс в графическом виде и пустить вдоль горизонтальной оси время, отложить на ней циклы, фазы, итерации и milestones, а вдоль вертикальной оси статические аспекты процесса, как это предписано, то результат будет выглядеть следующим образом:



При итерационном подходе, каждая из фаз процесса разработки состоит из нескольких итераций, целью которых является последовательное осмысление стоящих проблем, наращивание эффективных решений и снижение риска потенциальных ошибок в проекте. В то же время, каждая из последовательностей действий по созданию ПО выполняется в течение нескольких фаз, проходя пики и спады активности.

Каждый цикл итерации проекта начинается с планирования того, что должно быть выполнено. Результатом выполнения должен быть значимый продукт. Заканчивается же цикл оценкой того, что было сделано и были ли цели достигнуты.

Rational Unified Process, как продукт, состоит из:

- Размещаемой на Web базы знаний, которая состоит из руководств, шаблонов, наставлений по использованию инструментальных средств, и которая может быть разбита на:

- Обширные руководства для всех членов коллектива разработчиков, для каждого временного интервала жизненного цикла ПО. Руководства представлены в двух видах: для осмысления процесса на верхнем уровне, и в виде подробных наставлений по повседневной деятельности. Руководства опубликованы в HTML формате.
- Наставления по пользованию инструментальными средствами, которые автоматизируют большие разделы процесса создания ПО. Наставления опубликованы в HTML формате.
- Примеры и шаблоны для Rational Rose, которые служат руководствами по тому, как структурировать информацию в Rational Rose при следовании указаниям RUP.
- Шаблоны для SoDa – более десятка шаблонов для SoDa, которые помогают автоматизировать документирование ПО.
- Microsoft Word шаблоны – более 30 шаблонов, которые предназначены для поддержки документации по всем последовательностям действий и интервалам жизненного цикла ПО.
- Планы в формате Microsoft Project – для тех, кому трудно сразу перейти к созданию планов - отражают итерационную разработку. Данные документы помогают произвести такой переход.
- Development Kit – описывает то, каким образом можно конфигурировать и расширить RUP для специфических нужд проекта, и обеспечивает инструменты и шаблоны, помогающие это выполнить.
- Доступ к Resource Center, который содержит последние публикации, обновления, подсказки, методики, а также ссылки на add-on и сервисы.
- Книга Ph. Kruchten - Rational Unified Process-An Introduction. Книга содержит 277 страниц и является хорошим вступлением и обзором к процессу и базе знаний.

Сюда же – из книги «Профессиональное программирование. Системный подход» - классификация процессов разработки.

Сюда же – про эволюционное прототипирование !

Тема 16 - Классификация средств разработки. Визуальные средства быстрой разработки (RAD) – преимущества и недостатки.

План :

1. Классификация по книге (с примерами типов приложений)
2. RAD средства. Преимущества и недостатки
3. Средства поддержки командной разработки (CVS, Team Suite и пр.)

Моя классификация :

1. По полноте охвата : всеохватные, направленные на одно-два дела;
2. По универсальности : процесс-ориентированные, для разных процессов и пр.
3. По функциям (крупно) :
   3.1. Средства создания кода (редакторы, среды, генераторы);
   3.2. Средства моделирования и проектирования;
   3.3. Трансляторы и средства сборки.
   3.4. Средства отладки
   3.5. Средства моделирования требований
   3.6. Средства создания ресурсов и проектирования польз. интерфейса
   3.7. СУБД
   3.8. Средства планирования и поддержки проектов
   3.9. Средства оценки и повышения производительности
   3.10. Средства тестирования (модульного и приемочного)
   3.11. Средства рефакторинга
   3.12. Средства реверс. инжиниринга и документирования (в т.ч. - форматирование текста)
   3.13. Среды исполнения
   3.14. Библиотеки
   3.15. Средства развертывания (инсталляторы).
   3.16. Средства поддержки командной разработки.
   3.17. Различные утилиты
4. По предоставляемому интерфейсу : командной строки, текстовые, графические, многооконные, web.
5. По сетевым возможностям – однопользовательские, сетевые и др.
6. По степени автоматизации выполняемых действий : неавтоматич., RAD, генераторы. ...
   Популярность сред разработки – см. журнал !

Тема 17 - Защищенное программирование (некоторые тезисы) – см. журнал

План :

1. Помнить о безопасности (!)
2. Опасность – слишком доверяем вводу пользователя.
3. Примеры опасностей.
4. Перехват исключений и избавление от старых небезопасных функций С.
5. Хакерам всегда легче. Что делать – все, что не разрешено – запрещено !!!
   и т.д. (коротко).

Тема 18 – Неформальные критерии качества (должен был быть обзор книги Совершенный код)

План **:**

1. От себя : элементы стиля и мое видение качества.
2. По разделам книги.
3. Вместо заключения ...
   Вместо заключения (упорядочить !)

1. Простота, простота и еще раз простота ! (а) программы пишутся для людей; б) нам это никогда не понадобится !);
2. Программы должны быть самодокументируемыми и читабельными ! (для людей пишутся !) - Следите за стилем !
3. Разделяй и властвуй ! Модульность, классы, методы, слои и пр. Монолитность и длинные методы – главная беда !
4. Повторное использование. (Поэтому – см. п. 3). Не повторяйтесь в самом проекте и в родственных проектах. Делайте одну работу один раз !
5. Программы пишутся для заказчиков (общайтесь с ними !!!)
6. Итеративность и планирование – не давайте вашим проектам растягиваться на годы ! (см. п. 5 тоже).
7. Используйте готовые решения (паттерны). Не изобретайте велосипед ! Кроме того, паттерны дают общепринятые названия вашим решениям !
8. Защищайтесь от изменений ! (программа должна работать и после того, как вы ее развернули !) Главное – это способность вносить в нее изменения (быстро, качественно, и вы сами, и другие). Вам помогут паттерны и абстракции !
9. Пользуйтесь автоматическими модульными тестами !!!
10. Не бойтесь править и рефакторить код ! Не запускайте его (мойте посуду) ! Например – не допускайте дублирования !!! Тесты не дадут вам ошибиться (п. 9) ...
11. В серьезных и долгоживущих пакетах для повторного использования ключевую роль играют абстракции предметной области ! Освобождайте себя от зависимостей, связанных с конкретными технологиями !
12. Не жалейте времени на моделирование, но и не затягивайте с этим ! Не допускайте паралича анализа !!! Начинайте работать, код многое покажет (код – это и есть проект !). Золотая середина между моделированием и разработкой...
13. Приступая к работе, определите архитектуру и план !
14. Планируйте и тестируйте постоянно ! (постоянство – тоже путь к успеху)
15. Не работайте сверхурочно ! (планомерное постепенное движение вперед вместо аврала).
16. Не бойтесь вычеркивать дела, которые Вы явно не успеете выполнить (будьте честными сами с собой и с заказчиком) – планируйте от достигнутого !
17. Используйте UML (общепринятый стандарт).
18. Не давайте средам RAD запутать себя !
19. Не увлекайтесь конкретными сиюминутными технологиями – они быстро меняются, а жизнь проходит... (дополнение к п. 18). Не гонитесь за модой – она переменчива !
20. Учитесь работать в команде – один в поле не воин ...
21. Не забывайте о важности интерфейсов ! Все сложные системы состоят из большого числа мелких модулей, сопрягаемых через интерфейсы.
22. Используйте наследование в крайнем случае. Рассмотрите возможность использовать делегирование. Наследуйте от интерфейсов !
23. Не забывайте о защите информации !
24. Лучше изучайте базовые языки и пользуйтесь их возможностями (учите мат. Часть !) Понимайте отличие языка и средства разработки на этом языке !

25. Не используйте устаревшие библиотеки и функции ! Опасайтесь сторонних библиотек с недокументированными возможностями !
26. Используйте стандарты ! Чем шире распространен стандарт, тем лучше ! (например, используйте STL – это часть стандарта C++ !!!).
27. Читайте книги !
28. Пишите программы !!! :) Удачи !