

# Лекция 1: Об архитектуре

Юрий Литвинов

yurii.litvinov@gmail.com

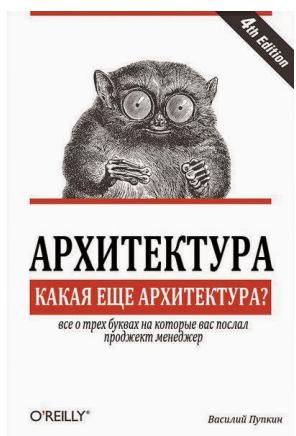
## 1. Введение

В этом курсе речь пойдёт про архитектуру программного обеспечения. Первое, что нужно сделать — это определиться, что именно мы называем архитектурой, и, собственно, этому и будет посвящена первая лекция. Тут оказывается, что не всё так просто, и все понимают под словом «архитектура» что-то своё. Формального определения понятия «Архитектура ПО» не существует. В академической среде под архитектурой понимают больше средства формального доказательства некоторых «архитектурных» свойств системы, исследуют формальные языки описания архитектуры и математические формализмы, с этим связанные (например, CSP<sup>1</sup>,  $\pi$ -исчисление). В индустриальной среде как правило даже не знают, что так можно, и понимают под архитектурой «совокупность важнейших решений об организации программной системы» (©Википедия) — декомпозицию системы на компоненты и способы взаимосвязи между компонентами. Так что мы начнём несколько издалека.

## 2. Архитектура? Какая архитектура?

### 2.1. О программе и программном продукте

Вообще, об архитектуре программного обеспечения заговорили только в начале 1990-х годов, что, конечно, уже прошлый век, но всё равно удивительно недавно — люди как-то программировали компьютеры с 1940-х годов, а UNIX, Windows, C++, Pascal и т.д. появились тогда, когда архитектуры как отдельной области программной инженерии ещё по сути не существовало. Многие профессиональные разработчики до сих пор «не верят» в архитектуру, многим студентам тоже кажется, что программирование — это больше про «фигачить код», а архитектура — это что-то ненужное, чем занимаются в огромных неповоротливых организациях. Разберёмся, почему.



<sup>1</sup> Communicating Sequential Processes



Рис. 1: Программа и системный программный продукт

На рисунке 1 воспроизведена знаменитая иллюстрация из ещё более знаменитой книги Фредерика Брукса «Мифический человеко-месяц, или Как создаются программные системы»<sup>2</sup>. В левом верхнем углу «Программа» — это кусок кода, который запускается и делает то, что нужно, на машине разработчика и если входные данные корректны и ничего плохого не происходит (работает так называемая «основная ветка» программы). Иногда, потому что никто не удосужился хорошошенько его протестировать или попытаться доказать корректность, так что на некоторых, даже вполне корректных входных данных оно падает или даёт неверный результат. Это то, что многие студенты и иногда даже, к несчастью, профессиональные разработчики, понимают под результатом деятельности программиста, и именно возможностями создания такого рода программ измеряют свою производительность и профпригодность. Есть соревнования, наподобие ACM ICPC Contest, в которых оцениваются подобные навыки, и эти соревнования весьма престижны. Там задумываться об архитектуре не нужно и, как правило, даже вредно.

Проблема в том, что «Программы» в этом смысле никому не нужны. Умение создавать такого рода программы необходимо для программиста, но это далеко не всё, что от квалифицированного программиста требуется. Люди готовы платить деньги не за код, который работает у разработчика, а за код, который работает у них. Они хотят доверять программе в том плане, что программа не падает на каждый чих и, скорее всего, не испортит их данные. Они хотят понимать программу — знать, как ей пользоваться, как её устанавливать и настраивать. Они хотят, чтобы программа работала с адекватной скоростью на их устройствах. Они хотят от программы чего-то нового, поэтому они хотят, чтобы разработчики не забрасывали программу, а продолжали поддерживать её, добавлять в неё новые возможности и править ошибки. Таким образом, ценность представляет не «программа», а «программный продукт», то есть та же программа, но оттестированная (покрытая модульными

<sup>2</sup> Брукс Ф. Мифический человеко-месяц или как создаются программные системы. – СПб : Символ-плюс, 2010, 304 с.

Простая игра для iOS	10000 LOC
Unix v1.0 (1971)	10000 LOC
Quake 3 engine	310000 LOC
Windows 3.1 (1992)	2.5M LOC
Linux kernel 2.6.0 (2003)	5.2M LOC
MySQL	12.5M LOC
Microsoft Office (2001)	25M LOC
Microsoft Office (2013)	45M LOC
Microsoft Visual Studio 2012	50M LOC
Windows Vista (2007)	50M LOC
Mac OS X 10.4	86M LOC

Рис. 2: Количество строк кода в типичных программных проектах

и интеграционными тестами), документированная (имеется пользовательская и техническая документация), с налаженным процессом сопровождения (наложен процесс исправления ошибок и выпуска новых версий), работающая на всех возможных входных данных и корректно обрабатывающая ошибочные ситуации. Брукс утверждает, что программный продукт примерно втрое более трудоёмок, чем программа, но на самом деле программный продукт может оказаться ещё дороже (одни только модульные тесты — обычно количество строк кода в них сопоставимо с количеством строк кода продукта, а часто и превосходит).

Кроме того, очень многие практические полезные программы сейчас работают не сами по себе, а в составе программных комплексов, будь то отдельные сервисы в распределённых приложениях, утилиты в операционных системах, плагины в средах разработки и так далее. Современная корпоративная информационная система может состоять из десятков различных приложений, развёрнутых на десятках различных вычислительных узлов, и все они должны быть интегрированы в единую систему. Это влечёт дополнительные трудозатраты на описание интерфейсов и протоколов общения между приложениями, и на интеграционное тестирование, сложность которого вполне может расти экспоненциально в зависимости от количества интегрируемых приложений. Ну а действительно полезный результат работы — это системный программный продукт, обладающий всеми описанными выше свойствами.

Таким образом, оказывается, что если требования к ПО невелики, его можно просто «сесть и написать», но в современном мире требования к ПО велики, поэтому часто приходится ещё и думать о разных других характеристиках ПО, помимо функциональности. Собственно, архитектура в этом сильно помогает. Ещё насчёт «сесть и написать» есть проблема в том, что ПО может быть слишком большим для этого. Немного примеров приведено в таблице 2<sup>3</sup>.

Архитектура — это как раз то, что в принципе позволяет создавать ПО размером больше 100000 строк кода. Без грамотной архитектуры большой проект рискует развалиться под собственной сложностью ещё в процессе разработки.

Есть довольно большой сегмент рынка ПО, связанный с разработкой мобильных приложений или одностраничных веб-сайтов без сложной логики, такие приложения, как пра-

<sup>3</sup> по данным <http://www.informationisbeautiful.net/visualizations/million-lines-of-code/>

вило, небольшие, так что при их разработке можно обходиться без архитектурных навыков. Многие разработчики всю жизнь работают junior- или mid-developerами, получают задачи уже декомпозиционными, и тоже обходятся без того, про что будет рассказываться в этом курсе. Все остальные вынуждены знать и уметь странные вещи типа паттернов, архитектурных стилей, UML и т.д., о чём и пойдёт речь далее.

## 2.2. Что такое архитектура и зачем она нужна

Общепринятого определения и даже понимания понятия «Архитектура» не существует, но на интуитивном уровне архитектура — это набор важнейших решений об организации программной системы — того, из каких компонентов она состоит, как эти компоненты взаимосвязаны друг с другом и с окружением, какие ограничения действуют на сами компоненты и взаимосвязи между ними, мотивация принятых решений. Стандарт ISO/IEC 42010 "Systems and software engineering — Architecture description" определяет архитектуру как "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution". Поскольку программные системы, как правило, очень сложны, их архитектура разделяется на части, каждая из которых описывает систему с какой-то определённой точки зрения (по аналогии с архитектурой зданий, где есть поэтажный план здания, план электропроводки, план водопровода и т.д.), эти части называются архитектурными видами. При этом каждый вид может состоять из нескольких описаний (неформальных, полуформальных или на каком-нибудь формальном языке), и эти описания могут различаться по уровню детализации (опять-таки, проводя аналогию с архитектурой зданий, может быть модель здания целиком и детальный план каждого помещения). В отличие от архитектуры здания, архитектура программной системы меняется и эволюционирует вместе с самой системой — ошибочные представления об архитектуре как деятельности, предшествующей программированию, я надеюсь, будут развеяны по ходу курса. Интересно то, что архитектура у системы есть всегда, даже если о ней никто не думал и никак её не описывал, просто она может оказаться не очень хорошей.

Для чего тратить усилия на явное описание архитектуры системы?

- С создания первого варианта архитектуры начинается реализация системы, это та деятельность, которая даёт понять, что и примерно как нужно писать. Продуманная архитектура может в разы сократить затраты на кодирование.
- Архитектура — это инструмент управления проектом. Даже оценить проект (а следовательно, принять решение, браться за него или нет) невозможно без декомпозиции задачи на подзадачи и выделения основных компонентов, крупнозернистая декомпозиция может определить состав команды и даже структуру организации, разрабатывающей систему. И по ходу выполнения проекта надо иметь чёткое представление о компонентах системы и связях между ними, чтобы понять, на каком этапе разработки мы сейчас находимся.
- Архитектура — это средство обеспечения переиспользования, как третьесторонних компонентов, так и переиспользования внутри системы (или группы систем). Именно архитектура позволяет выделить возможные кандидаты на переиспользование, определить требования к третьесторонним компонентам, определить критерии

выбора, выделить и инкапсулировать функциональность своих компонентов, чтобы сделать их переиспользуемыми.

- И главное, явное описание архитектуры системы позволяет делать выводы о некоторых её качествах ещё до того, как написана первая строчка кода. Несколько разных вариантов архитектуры может быть разработано и проанализировано, среди них может быть выбран вариант, наиболее подходящий для текущей ситуации и только после этого реализован. Формальные средства описания архитектуры позволяют иногда получать не только качественные, но и количественные характеристики системы, например, оценивать ожидаемую производительность или требуемую память. К сожалению, такие средства не очень распространены в промышленности, так что и в этом курсе мы их почти не коснёмся.

Интересно, что заказчику архитектура системы совершенно не интересна, они ожидают, что система будет делать то, что им нужно, причём хорошо и быстро. Как она устроена внутри, им особо не интересно. Как писал Алан Купер<sup>4</sup>, если бы в магазинах продавали дырки в стене, никто не покупал бы дрели.

## 2.3. Профессия «Архитектор»

В крупных проектах архитектор — это, как правило, специально выделенный человек или даже группа людей, в задачи которых входит разработка и описание архитектуры системы, доведение её до всех заинтересованных лиц, контроль реализации архитектуры и поддержание её в актуальном состоянии по ходу разработки и сопровождения проекта. Есть профессиональный стандарт «Архитектор программного обеспечения», диплом «Математическое обеспечение и администрирование информационных систем» позволяет вести профессиональную деятельность и по этой профессии (почему, собственно, курс по архитектуре есть в программе подготовки). Стандарт определяет цель деятельности архитектора так:

Создание и сопровождение архитектуры программных средств, заключающейся

- в синтезе и документировании решений о структуре;
- компонентном устройстве;
- основных показателях назначения;
- порядке и способах реализации программных средств в рамках системной архитектуры;
- реализации требований к программным средствам;
- контроле реализации и ревизии решений

Вот некоторые трудовые функции архитектора согласно этому же стандарту:

- Создание вариантов архитектуры программного средства
  - Определение перечня возможных типов для каждого компонента

<sup>4</sup> А. Купер, Р.М. Рейманн, Д. Кронин, Алан Купер об интерфейсе. Основы проектирования взаимодействия, Символ-Плюс, 2009г, 688 С.

- Определение перечня возможных архитектур развертывания каждого компонента
  - Определение перечня возможных слоев программных компонентов
  - Определение функциональных характеристик и возможностей, включая эксплуатационные, физические характеристики и условия окружающей среды, при которых будет применяться каждый компонент
  - Определение перечня возможных протоколов взаимодействия компонентов
  - Определение перечня возможных механизмов авторизации
  - Определение перечня возможных механизмов аутентификации, поддержки сеанса
  - Определение перечня возможных схем кеширования
  - ...
- Документирование архитектуры программных средств
    - Разработка документации программных средств в своей части
    - Поддержка изменений в документации
  - Реализация программных средств
    - Анализ качества кода: анализ зависимостей; статический анализ кода
    - Испытания создаваемого программного средства и его компонентов
    - Технические и управленческие ревизии создаваемого программного средства
  - Оценка требований к программному средству
  - Оценка и выбор варианта архитектуры программного средства
  - Контроль реализации программного средства
  - Контроль сопровождения программных средств
  - Оценка возможности создания архитектурного проекта
  - Утверждение и контроль методов и способов взаимодействия программного средства со своим окружением
  - Модернизация программного средства и его окружения

Как видим, архитектор, помимо собственно разработки архитектуры, выполняет также техническое руководство проектом, контролирует ход его реализации, оценивает качество результата на всём протяжении разработки и даже сопровождения. Таким образом, архитектор — это ключевой технический специалист в команде. Кроме того, на архитектора же чаще всего возлагают обязанности по коммуникации между разработчиками и руководством, часто архитектор принимает участие в общении с заказчиком, поэтому для архитектора важны не только технические, но и социальные навыки. Хороший архитектор должен обладать большим кругозором и уметь быстро погружаться в предметную область, в которой разрабатывается проект — в отличие от программиста, которому обычно достаточно разбираться в той технологии, которую он использует для реализации. Картинка,

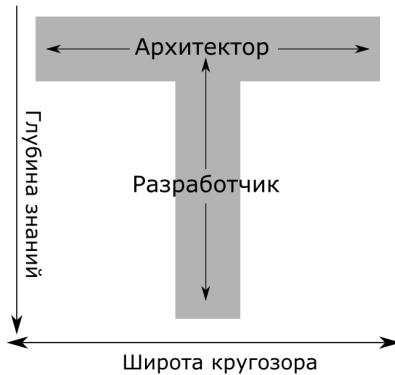


Рис. 3: Знания архитектора и знания разработчика

иллюстрирующая соотношение знаний архитектора и разработчика, приведена на рисунке 3.

Часть знаний и умений архитектора может прийти только с опытом, поэтому обычно будущему архитектору сначала приходится поработать на позиции разработчика. В небольших командах часто архитектор вынужден также выполнять функции разработчика, а часто и наоборот, разработчики вынуждены заниматься архитектурой помимо, собственно, кодирования. В общем-то, ничего страшного в этом нет, в отличие от строительства зданий в мире разработки ПО не удается достичь чёткого разделения труда как между архитектором и строителем (ведь программа по сути — это проект, строит её компилятор), так что функции архитектора и разработчика перекрываются. Более того, если архитектор сидит «в башне из слоновой кости» и только говорит разработчикам, что делать, он быстро теряет связь с кодом и утрачивает понимание реализации. Это приводит к тому, что его архитектурные решения игнорируются разработчиками или, если архитектор настаивает, приводят к ухудшению качества системы. Поэтому хороший архитектор должен быть прежде всего опытным программистом и писать код вместе с командой (может быть, уделяя больше внимания интерфейсам компонентов, протоколам связи и интеграционным тестам, но всё-таки код).

Но при этом архитектор не должен заниматься микроменеджментом и пытаться навязывать программистам своё видение реализации. Если задача декомпозирована до уровня конкретных методов классов, программистам будет неинтересно работать в таком проекте и они уйдут. Архитектору следует доверять программистам и оставлять реализационные аспекты им. По моему мнению, граница между ответственностью архитектора и ответственностью программиста проходит где-то на уровне паттернов проектирования из известной книжки Э. Гаммы<sup>5</sup>. С паттернами любой программист должен быть в состоянии разобраться самостоятельно, поэтому какие паттерны лучше использовать для реализации компонента — уже скорее дело программиста, а вот что должен делать компонент и как он должен быть связан с остальными — дело архитектора. Но паттерны проектирования в этом курсе всё равно будут, потому что это важная штука.

<sup>5</sup> Must read: Гамма Э. и др. Приемы объектно-ориентированного проектирования. – Издательский дом "Питер 2016, 366С.

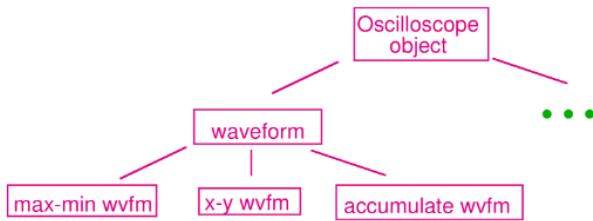


Рис. 4: Объектно-ориентированная модель осциллографа

### 3. Пример: ПО для осциллографа

Посмотрим на примере, какое влияние архитектура оказывает на приложение и что примерно ожидается от архитектора. Рассмотрим некоторую компанию, которая занимается проектированием осциллографов — да-да, устройств для записи параметров электрического сигнала<sup>6</sup>. Если раньше осциллографы были аналоговыми и никакого софта к ним было вообще не нужно, то сейчас осциллограф умеет считывать кучу параметров, оцифровывать их, сохранять во внутреннем хранилище, выполнять разные фильтрации и преобразования (например, преобразование Фурье), отображать результаты на экране (с тач-скрином, меню и встроенной справкой) и выдавать в сеть, где этими результатами могут пользоваться как обычные компьютеры, так и другие устройства. Эта наша компания выпускает сразу несколько разных моделей осциллографов, да ещё и предоставляет услугу настройки своей продукции под конкретные нужды конкретного клиента. Последнее раньше было просто (просто меняем прошивку осциллографа на слегка подправленную), но с ростом функциональности приборов стало сложно и дорого. Поэтому понадобилась Архитектура.

В первой версии архитектуры авторы пошли по пути создания объектно-ориентированной модели предметной области (как мы увидим далее, это наиболее распространённый и считающийся идеологически правильным сейчас подход, но авторы об этом не знали, 1993 год же). Были выделены основные сущности, определены связи между ними, нарисованы диаграммы в духе диаграммы из рисунка 4<sup>7</sup>.

Это не дало желаемого результата, потому что хоть все сущности и были аккуратно выделены, их оказалось слишком много, а разбить сущности на модули, компоненты или слои, объектная модель не помогала. Кроме того, было неочевидно, как разбить функциональность по объектам (например, измерения должны моделироваться как часть объектов, представляющих данные, которые измеряются, или как отдельные объекты). Сейчас все эти проблемы объектной модели более-менее известно, как решать, но тогда авторы решили отказаться от такого подхода и попробовать что-то ещё. Впрочем, результаты не пропали даром, объектная модель всё равно пригодилась, и главное, улучшила понимание предметной области архитекторами.

Следующая итерация была уровневой архитектурой: система разделялась на слои, самый нижний отвечал за взаимодействие непосредственно с оборудованием, на нём стро-

<sup>6</sup> Пример и всё изложение в этом разделе позаимствованы из статьи Garlan D., Shaw M. An introduction to software architecture //Advances in software engineering and knowledge engineering. – 1993. – Т. 1. – №. 3.4.

<sup>7</sup> Оригинальная иллюстрация из всей той же статьи An introduction to software architecture

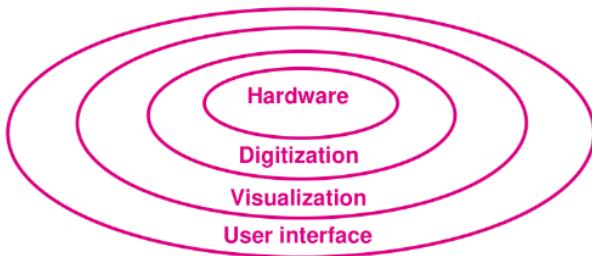


Рис. 5: Уровневая модель осциллографа

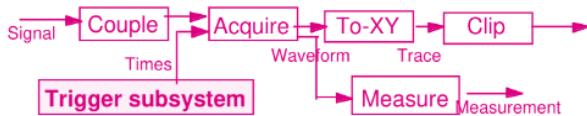


Рис. 6: Модель осциллографа, основанная на потоке данных

ился слой оцифровки и слой цифровой обработки, на нём слой визуализации, на нём — слой пользовательского интерфейса, и, как это принято в уровнях архитектурах, каждый слой мог общаться только со слоем непосредственно выше или ниже его самого. Получилось что-то наподобие рисунка 5<sup>8</sup>.

Такая модель (заметим, что внутри слоёв вполне могли использоваться объекты из предыдущей) была проста для понимания и хорошо структурировала всю систему. Однако она тоже оказалась не очень, потому что на самом деле все слои должны были знать про все остальные слои, например, пользовательский интерфейс должен был позволять настраивать параметры оцифровки и различных преобразований, в виде графиков могли отображаться не только результаты хитрых вычислений, но и «сырой» сигнал. Все преимущества слоистого подхода при таком положении вещей терялись, поэтому авторы продолжали работать над архитектурой получше.

Следующая модель — «каналы и фильтры» — модель, рассматривающая функциональность осциллографа как набор последовательных преобразований над данными. Концепция такой архитектуры изображена на рисунке 6<sup>9</sup>.

Теперь ничто не мешает пользовательскому интерфейсу показывать «сырой» сигнал, к тому же такая точка зрения на систему близка к точке зрения пользователей осциллографа — инженеров. Однако пользовательские настройки всё ещё не очень хорошо ложились в эту модель, потому что если пользовательский интерфейс мог быть только в конце «трубопровода», то получалось ещё хуже, чем в слоистой модели, когда пользователь на самом деле не мог ничего менять, а если интерфейс мог быть повсюду, получалась каша хуже той, что была в чисто объектной модели.

Поэтому архитектура был слегка модифицирована, фильтрам добавили возможность принимать два разных типа данных — данные, с которыми фильтры, собственно, работали

<sup>8</sup> Тоже иллюстрация из статьи

<sup>9</sup> Опять-таки, иллюстрация из статьи

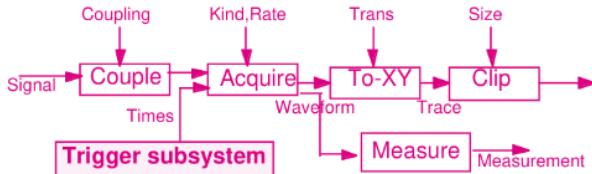


Рис. 7: Модель осциллографа, основанная на потоке данных с управляющими входами

и передавали дальше, и управляющие сигналы, см. рисунок 7.

Теперь всё было хорошо, настройки моделировались как управляющие входы, отображаться могло всё, что угодно, и главное, система была легко расширяема и конфигурируема — достаточно было описать новый тип фильтра, описать его связи с существующими фильтрами, и так получить новую функциональность для осциллографа. Так что на этой (точнее, на немного улучшенной, но по сути этой) архитектуре авторы и решили остановиться.

### 3.1. Выводы

Из этого примера видно, что мы можем делать какие-то утверждения о свойствах разрабатываемой системы, базируясь исключительно на её структурных свойствах, не написав ни строчки кода и даже не выбрав язык реализации. Причём, структура системы оказывает очень сильное влияние на весь ход проекта: в первом варианте, с чистой объектно-ориентированной моделью система оказывалась несопровождаемой и нерасширяемой в долгосрочной перспективе, во втором варианте, со слоями, и в третьем, с каналами и фильтрами система не могла красиво с точки зрения реализации удовлетворить функциональным требованиям. Естественно, можно было во время реализации «затолкать» в архитектуру нужную функциональность, например, «пробросить» данные и параметры конфигурации через несколько слоёв, сделав функции, просто делегирующие вызов на слой ниже, но это опять-таки удорожило бы разработку и сопровождение.

Ещё из примера видно, что все наши рассуждения носят весьма оценочный и субъективный характер. Это тоже весьма типично для архитектуры, есть формальные методы, позволяющие получить количественные результаты, но всё равно, большая часть принимаемых решений не может быть объективно оценена. Архитектура — это во многом гуманистическая область, архитектура системы создаётся для людей и предназначена прежде всего для упрощения понимания людьми структуры системы. Архитектура должна объяснять и упрощать, а эти вещи сложно оценить и сложно сравнивать. Поэтому часто оказывается, что всей команде приходится полагаться на интуицию и чувство вкуса архитектора. Из этого же следует, что не бывает «правильной» или «неправильной» архитектуры. Какой бы ужас вы ни породили, скорее всего, его удастся ценой некоторых усилий реализовать так, чтобы он таки заработал. Тем не менее, бывают архитектуры явно удачные и явно не удовлетворяющие требованиям, однако отличить одну от другой заранее очень сложно (особенно если требования заранее неизвестны, как чаще всего и бывает). Страшного в этом ничего нет, потому что архитектура меняется во время жизни приложения, так что ошибки можно исправить. Но это может стоить довольно дорого, например, архитек-

турный рефакторинг проекта QReal:Robots<sup>10</sup> потребовал примерно человека-года усилий. Причём надо помнить, что во время архитектурного рефакторинга добавлять полезную функциональность и вообще развивать продукт не получится.

Третье наблюдение состоит в том, что архитектурные подходы, которые мы применяли, не специфичны для осциллографов. Уровневое разбиение, которое тут нам не подошло, вполне успешно применяется в сетевых протоколах (например, модель OSI — типичная уровневая архитектура), объектно-ориентированная модель предметной области — основа структуры типичных информационных систем и заодно основа их схемы баз данных. Собственно, обобщения таких подходов называются архитектурными стилями, это что-то вроде архитектур архитектуры — архитектурный стиль определяет основные структурные элементы конкретной архитектуры и ограничения на них. Архитектура у каждой системы своя, а вот известных архитектурных стилей не так уж много, поэтому любой адекватный архитектор должен их знать. Поэтому у нас в курсе будет отдельная пара с обзором стилей и несколько пар с их подробным обсуждением.

Четвёртый вывод из примера состоит в том, что систему можно рассматривать с разных точек зрения. Первый вариант архитектуры, с объектами, концентрировался на статической структуре системы и описывал сущности, из которых система состоит. Третий вариант, с фильтрами, концентрировался на потоках данных, которые существуют в системе, их преобразованиях и взаимосвязях. Также можно заметить и разные уровни детализации — разбиение системы на слои явно более «крупнозернисто», чем разбиение на конкретные классы, кроме того, дальше не детализируется — нет никакого смысла разбивать слои на подслои и т.д. В какой-то момент каждый слой должен быть представлен моделью с классами, так что если бы слоистая архитектура нам подошла, она состояла бы минимум из двух разных видов описаний. Такие описания называют *архитектурными видами* (architectural view), а совокупность видов, рассматривающая систему единообразно — *архитектурной точкой зрения* (architectural viewpoint). Например, диаграмма классов и диаграмма «сущность-связь» — это два разных вида, относящиеся к одной точке зрения.

## 4. Архитектурное описание, архитектурные виды

По поводу архитектурных видов подробно написано в стандартах IEEE 1016-2009 «Software Design Descriptions» и ISO/IEC/IEEE 42010:2011 «Architecture description». IEEE 1016 выделяет аж 12 точек зрения на систему:

- Контекст — описывает, что система должна делать, фиксирует окружение системы. Состоит из сервисов и акторов, которые могут быть связаны информационными потоками. Система представляет собой «чёрный ящик». Корень иерархии уточняющих дизайн системы видов, стартовая точка при проектировании системы. Обычно описывается с помощью диаграмм активностей UML, IDEF0 (SADT).
- Композиция — описывает крупные части системы и их предназначение. Предназначен для локализации и распределения функциональности системы по её структурным элементам, impact analysis-a, облегчения переиспользования (в том числе, покупки компонентов), оценки, планирования, управления проектом, определения

<sup>10</sup> Образовательная среда программирования роботов,  
<https://github.com/qreal/qreal/tree/master/plugins/robots>

необходимой инструментальной поддержки (репозитории, трекер и т.д.). Обычно описывается с помощью диаграмм компонентов UML, IDEF0, Structure Chart.

- Логическая структура — структура системы в терминах классов, интерфейсов и отношений между ними. Типичные языки — диаграммы классов UML, диаграммы объектов UML.
- Зависимости — определяет связи по данным между элементами: разделяемая между элементами информация, порядок выполнения и т.д. Необходим для анализа изменений, идентификации узких мест производительности, планирования, интеграционного тестирования. Типичные языки — диаграммы компонентов UML, диаграммы пакетов UML.
- Информационная структура — определяет персистентные данные в системе (информация, которую требуется хранить, схема БД, доступ к данным). Типичные языки — диаграммы классов UML, IDEF1x, ER, ORM
- Использование шаблонов — документирование использования локальных паттернов проектирования. Типичные языки — диаграммы классов UML, диаграммы пакетов UML, диаграммы коллaborаций UML.
- Интерфейсы — специфицирует информацию о внешних и внутренних интерфейсах, не прописанную явно в требованиях. Пользовательский интерфейс рассматривается отдельным видом в рамках этой точки зрения. Типичные языки — IDL, диаграммы компонентов UML, макеты пользовательского интерфейса, неформальные описания сценариев использования.
- Структура системы — рекурсивное описание внутренней структуры компонентов системы. Типичные языки — диаграммы композитных структур UML, диаграммы классов UML, диаграммы пакетов UML.
- Взаимодействия — описывает взаимодействие между сущностями: почему, когда, как и на каком уровне выполняется взаимодействие. Типичные языки — диаграммы композитных структур UML, диаграммы взаимодействия UML, диаграммы последовательностей UML.
- Динамика состояний — описание состояний и правил переходов между состояниями в реактивных системах. Типичные языки — диаграммы конечных автоматов UML, диаграммы Харела, сети Петри.
- Алгоритмы — описывает в деталях поведение каждой сущности, логику работы методов. Типичные языки — диаграммы активностей UML, псевдокод, настоящие языки программирования.
- Ресурсы — описывает использование внешних ресурсов (как правило, аппаратных или третьесторонних сервисов). Типичные языки — диаграммы развёртывания UML, диаграммы классов UML, OCL.

Хорошая новость в том, что все эти архитектурные виды необязательны, стандарт требует только общие сведения о системе (назначение, границы системы, контекст, в котором система существует) и те виды, которые архитектор считает важными.

Можно обратить внимание, сколько раз в стандарте упоминаются UML и другие языки моделирования (что интересно, сам стандарт использует диаграммы классов UML для описания своих основных концепций и терминов). Модели вообще очень важны для архитектуры системы, потому что ПО само по себе очень сложно, а модели по определению упрощают сложное, оставляя существенное. К несчастью, модели ПО, в отличие от математических и физических моделей, принципиально не могут быть простыми, потому что сложность ПО — это то самое, с чем работает архитектура, от сложности нельзя абстрагироваться и объявить её несущественной. Но для этого, собственно, и придуманы виды — каждый вид рассматривает свой какой-то аспект системы, сложность каждого вида ограничена, хотя сложность всей системы может быть сколь угодно большой.

С языками визуального моделирования, в особенности с UML, мы ещё познакомимся подробно в этом курсе. UML — это общепринятый промышленный стандарт, так что уметь читать UML-диаграммы надо всем, даже тем, кто хочет всю жизнь проработать младшим разработчиком, а умение рисовать UML-диаграммы — обязательный навык для любого архитектора, даже если в каком-то конкретном проекте или организации ими не пользуются.

## 5. Роль архитектуры в жизненном цикле ПО

Есть распространённое заблуждение, что работа архитектора напрямую связана с фазой проектирования жизненного цикла ПО — архитектор получает проанализированные требования от аналитиков, рисует диаграммы, пишет design document, отдаёт всё, что получилось, программистам на реализацию и идёт заниматься следующим проектом. В некоторых компаниях, между прочим, так и делают, практика «кочующих архитекторов» довольно эффективна по стоимости, поскольку можно держать одну опытную архитектурную команду, обслуживающую сразу много проектов. Тем не менее, современная точка зрения на архитектуру больше склоняется к тому, что архитектура это непрерывная активность, находящаяся в центре процесса разработки, см., например, рисунок 8 из книги Z. Quin, “Software Architecture”<sup>11</sup>. Создание, поддержка и эволюция архитектуры — это такие же деятельности, необходимые на всех этапах жизненного цикла, как контроль версий или управление проектом, архитектура при этом ещё и направляет остальные деятельности.

Много кто пишет, что собирать требования надо безотносительно конкретной реализации, чтобы требования отражали потребности пользователя, а не то, что мы ему (иногда подсознательно) навязали исходя из своих возможностей по реализации. Тем не менее, сбор и анализ требований — это уже архитектурная активность, в этот момент рисуются первые модели системы, пишутся первые документы, которые лягут в основу будущей архитектуры, а затем и реализаций. Например, диаграмма случаев использования UML, пример которой представлен на рисунке 9, она показывает роли пользователей системы и доступную им функциональность. Нередко в сборе требований и общении с заказчиком участвует архитектор.

<sup>11</sup> Z. Quin, J. Xing, X.Zheng, “Software Architecture”, Zhejiang University Press, 2008, 337pp

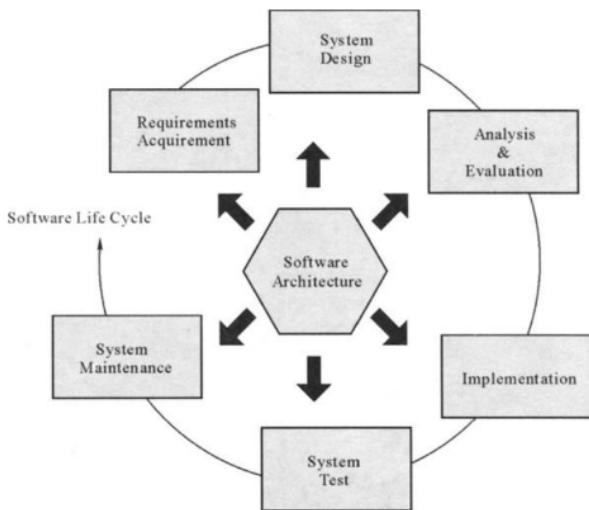


Рис. 8: Место архитектуры в жизненном цикле ПО

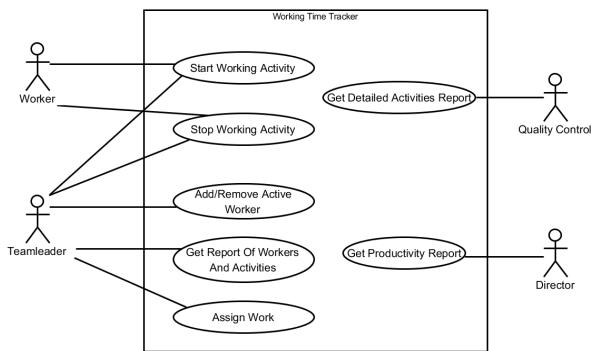


Рис. 9: Диаграмма случаев использования UML

Требования вообще чрезвычайно важны для архитектуры, поскольку это исходные данные для проектирования. Напомним, что требования бывают таких типов.

- Функциональные — то, *что* система должна делать.
- Нефункциональные — то, *как* система должна это делать.
  - Требования на эффективность по времени работы и по используемым ресурсам.
  - Требования на масштабируемость — насколько адекватно система справляется с ростом нагрузки.
  - Требования на удобство использования — насколько легко учиться пользоваться системой и насколько удобно собственно пользоваться ею.
  - Требования на надёжность.
    - \* Время наработки на отказ — сколько времени в среднем система работает без отказов.
    - \* Время восстановления после сбоя — за сколько времени систему можно восстановить, если она всё-таки отказалась. Заметим, что время наработки на отказ и время восстановления после сбоя — это разные вещи, имеющие разное значение в разных ситуациях. Например, для межпланетного зонда время восстановления после сбоя может быть вообще не интересно, если мы неправильно сориентировали антенну, то всё, он навсегда потерян в глубинах космоса, тогда как время наработки на отказ критично — если зонду лететь 20 лет, но в среднем раз в два года бортовое ПО падает с критической ошибкой, запускать такой зонд бессмысленно. Для сотового телефона наоборот, один пропущенный из-за сбоя вызов не страшен, если можно быстро перезвонить.
    - \* Корректность поведения при сбое (*failsafe*) — есть системы, в которых даже критическая ошибка должна приводить к корректному завершению работы (классический пример с ядерным реактором и с банковской системой).
  - Требования на безопасность.
  - Требования на сопровождаемость и расширяемость.
  - ...
- Ограничения
  - Технические — например, если вы программируете под iPhone, то ограничены в выборе инструментария.
  - Бизнес-ограничения — например, система должна быть сделана за месяц, иначе на уже назначенной пресс-конференции будет нечего показать.

Наибольшее влияние на архитектуру оказывают, как ни странно, нефункциональные требования. Дело в том, что написать работающую программу, в общем-то, не сложно, и это можно сделать миллионом различных способов. Сложно сделать так, чтобы она была

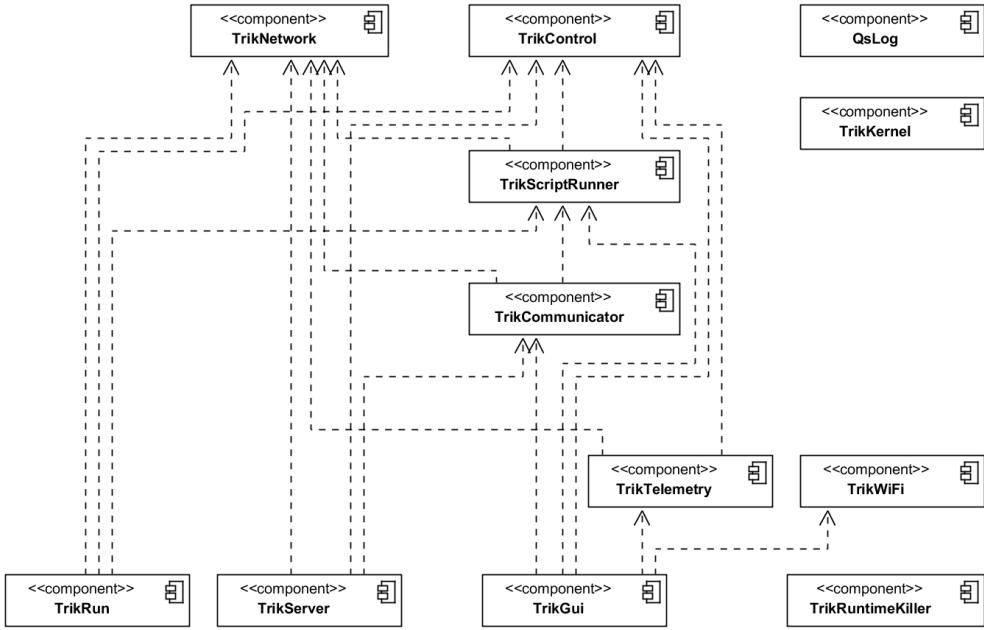


Рис. 10: Диаграмма компонентов UML

ещё и быстрой, сопровождаемой, надёжной и т.д. Поэтому и архитектурный стиль выбирается исходя из приоритетов нефункциональных требований, и архитектура проектируется в большой степени с их учётом.

На стадии проектирования архитектурные вопросы естественным образом оказываются в центре внимания. Обратите внимание, что сейчас мало где используется водопадная модель разработки, поэтому стадию проектирования проекты чаще всего проходят несколько раз (например, в начале каждой итерации в agile-методологиях). На фазе проектирования выполняется декомпозиция задачи, определяются границы компонентов, определяются интерфейсы и протоколы общения. На стадии проектирования же решаются вопросы, общие для всей системы: стратегия обработки ошибок, стратегия логирования, стратегия выкатывания обновлений и т.д. На стадии проектирования же создаётся большая часть архитектурной документации. Для этого опять-таки оказываются полезны визуальные языки, например, диаграммы компонентов UML, которые показывают «вид с высоты птичьего полёта» на систему. Например, высокоуровневая архитектура прошивки робота представлена в виде диаграммы компонентов на рисунке 10.

На фазах реализации, тестирования и поддержки за архитектурой необходимо следить, причём, как правило, делать это приходится вручную — архитектурные ограничения редко удаётся формализовать так, чтобы они могли проверяться инструментами. Архитектура может быть подвержена двум видам проблем. Первая из них — это *architectural drift* — появление в «фактической» архитектуре системы (которая, кстати, называется «*descriptive architecture*» — «описывающая архитектура») важных архитектурных решений, которых не было в разработанной архитектуре (которая, в свою очередь, называется «*prescriptive*

*architecture*» — «предписывающая архитектура»). Такие решения вполне могут ничего не ломать и быть вполне адекватными, но, как правило, оказываются незадокументированными, никто не задумывался об их «глобальных» последствиях, они применяются только к части системы, не становясь общим принципом, что, разумеется вносит свою долю хаоса. Вторая проблема — это *architectural erosion* — нарушение в фактической архитектуре системы принципов, заложенных в её предписывающей архитектуре. Поскольку архитектурные ограничения сложно проверять автоматически, архитектурная эрозия возникает на практике очень часто в процессе багфиксов, рефакторингов, добавления новой функциональности, и, опять-таки, даже если напрямую ничего не ломает, вносит хаос в систему (в гораздо большей степени, чем drift) и постепенно превращает систему в «большой ком грязи», где всё связано со всем и никто не знает, как оно работает. Архитектурная эрозия чем-то похожа на законы термодинамики и энтропию. Бороться с эрозией, так же, как и с энтропией, в общем-то, бесполезно — надо помнить, что каждый рефакторинг, каждый багфикс, каждая написанная новая строка кода имеет свою цену в виде потенциальной архитектурной эрозии и сползания проекта в хаос. Хорошая и чётко описанная архитектура, постоянное внимание архитектора и архитектурные рефакторинги могут сильно замедлить этот процесс.

Тем не менее, в жизненном цикле системы может наступить момент, когда внесение изменений становится слишком дорогим из-за сильной эрозии архитектуры. В это время может потребоваться *восстановление архитектуры* по имеющемуся коду, рефакторинг этой архитектуры и далее глобальный рефакторинг кода для того, чтобы привести его в соответствие новой архитектуре. В особо запущенных случаях может потребоваться применять методы реинжиниринга (например, если архитектурная документация вообще утеряна). В любом случае, процесс это сложный и долгий, причём, пока выполняется такая работа, никакое развитие системы невозможно.

## 6. Пример: Apache Hadoop

Хороший пример архитектурной эрозии и того, во что красавая на бумаге архитектура превращается при реализации — архитектура системы Apache Hadoop. Apache Hadoop — это широкоизвестный фреймворк для распределённой обработки больших данных. Основной принцип его работы — это паттерн «Map-Reduce», когда вычисление раскидывается на много независимых узлов, каждый из которых обрабатывает свой кусок данных, а потом то, что получилось, складывается на один узел, который объединяет результаты. Плюс в процессе участвует распределённая файловая система (HDFS, Hadoop Distributed File System), которая хранит и предоставляет вычислительным узлам данные. Hadoop, например, используется в Yahoo, масштабы развёрнутой системы там порядка 40000 серверов и 40 петабайт данных (по крайней мере, если верить The Architecture of Open Source Applications, том 1).

С точки научной зрения Hadoop интересен тем, что он относительно большой (порядка 200К значимых строк кода, порядка 1700 классов, на Java), с открытым исходным кодом и широко известен (часто используется в научных статьях как набор данных для разного рода техник анализа архитектуры). В статье J. Garcia, I. Krka, C. Mattmann, N. Medvidovic, “Obtaining Ground-Truth Software Architectures” Hadoop использовался как один из четырёх примеров систем, для которых авторы попытались восстановить их «описывающую

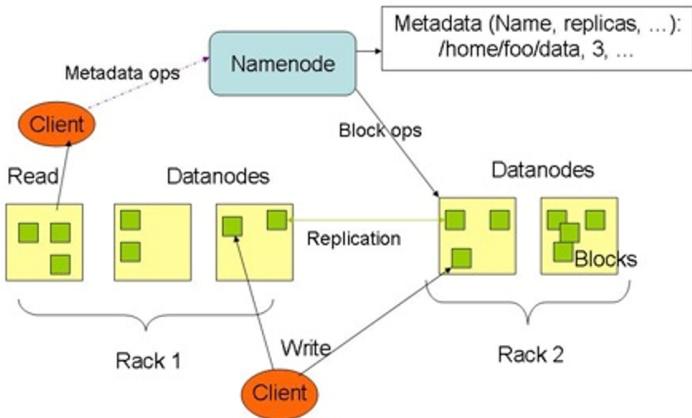


Рис. 11: HDFS as-designed.

архитектуру» по исходному коду, отдать её на рассмотрение разработчикам/архитекторам системы и получить тем самым более-менее точное описание того, как система устроена на самом деле. И, конечно, сравнить с «предписывающей архитектурой». В работе участвовало два аспиранта, на это у них ушло порядка 120 рабочих часов, потом ещё 8 часов инженера из Yahoo, коммитившего в Hadoop, который должен был посмотреть на адекватность результатов и высказать замечания.

Вот что получилось. На рисунке 11 показана предписывающая архитектура Apache Hadoop (по крайней мере её часть, относящаяся к файловой системе, HDFS), как она описана в архитектурной документации<sup>12</sup>. Кластером управляет NameNode, который хранит метаинформацию о файлах и управляет доступом клиентов к файлам. Сами файлы хранятся не целиком, а блоками, блоки разделены по DataNode-ам и реплицированы на другие DataNode-ы, чтобы если один вышел из строя, данные не потерялись. Клиент получает у NameNode информацию о том, где его данные, и потом взаимодействует с DataNode-ами напрямую.

На рисунке 12 показана архитектура той же HDFS, восстановленная по исходникам (и исходная архитектура для сравнения). Диаграмма намеренно не очень читаема, её основной смысл в том, что реальность гораздо сложнее и состоит из значительно большего количества структурных компонентов и связей. На самом деле, тут есть некоторый обман, потому что рисунок 11 показывает экземпляры компонентов системы во время выполнения, тогда как рисунок 12 показывает статическую структуру системы, но авторам пришлось работать с той архитектурной документацией, что есть. Описания статической структуры системы в документации вообще не нашлось (даже с учётом того, что про HDFS есть глава в книге *The Architecture of Open Source Applications*).

Если посмотреть на Hadoop в целом (HDFS + MapReduce), то получится статическая структура, изображённая на рисунке 13. Зелёным, как и раньше, выделены компоненты, относящиеся к HDFS, жёлтым — к MapReduce. На диаграмме не показаны различные утилиты.

<sup>12</sup> URL: <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>

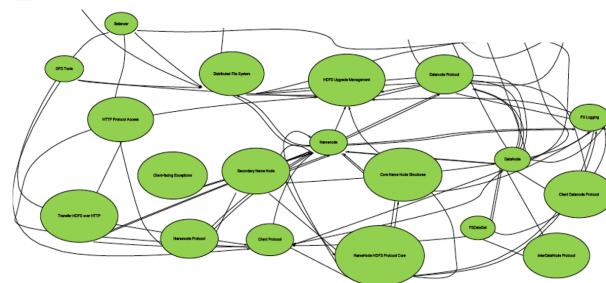
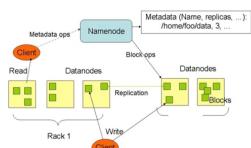


Рис. 12: HDFS as-built.

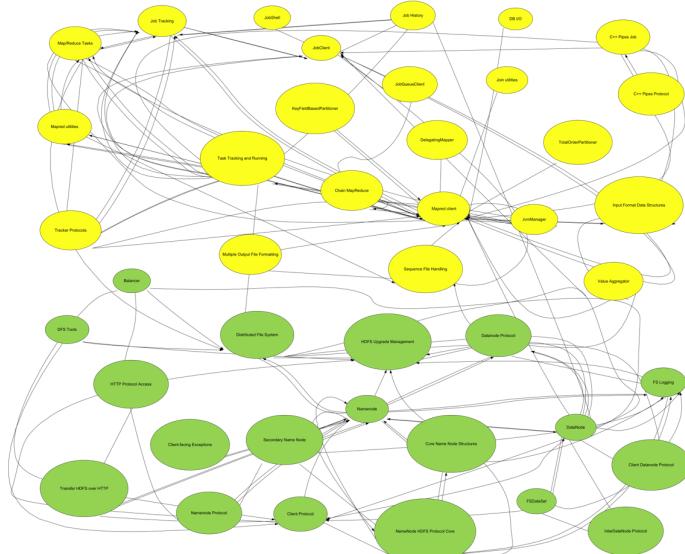


Рис. 13: HDFS + MapReduce as-built.

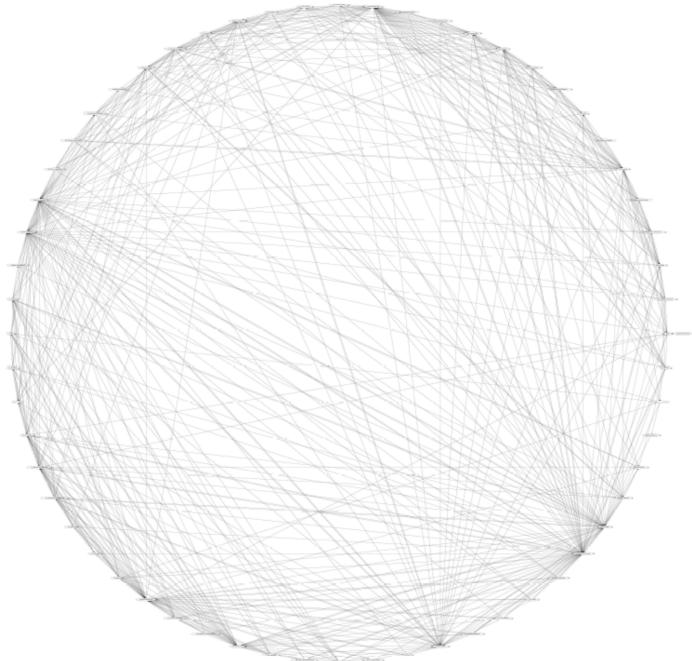


Рис. 14: Диаграмма зависимостей компонентов Hadoop.

А на рисунке 14 можно видеть диаграмму, показывающую все выделенные компоненты и их зависимости.

Или вот ещё один вид на восстановленную архитектуру, с иерархическим размещением компонентов, рисунок 15. Опять-таки, на рисунке намеренно не видны детали.

Интересно, что в ходе исследования выяснилось, что архитектурные компоненты системы не всегда соответствуют структуре папок исходников. В случае с Hadoop было выделено 68 отдельных компонентов, из которых 21 были утилиты, а 47 реализовывали бизнес-логику. 18 из 68 компонентов имели куски реализации в разных пакетах, 40 компонентов имели в «своих» пакетах куски реализации других компонентов, и в общей сложности 58 из 68 компонентов не соответствовали структуре пакетов в исходниках. Так что просто посмотреть на структуру папок программы совершенно недостаточно, чтобы восстановить её архитектуру.

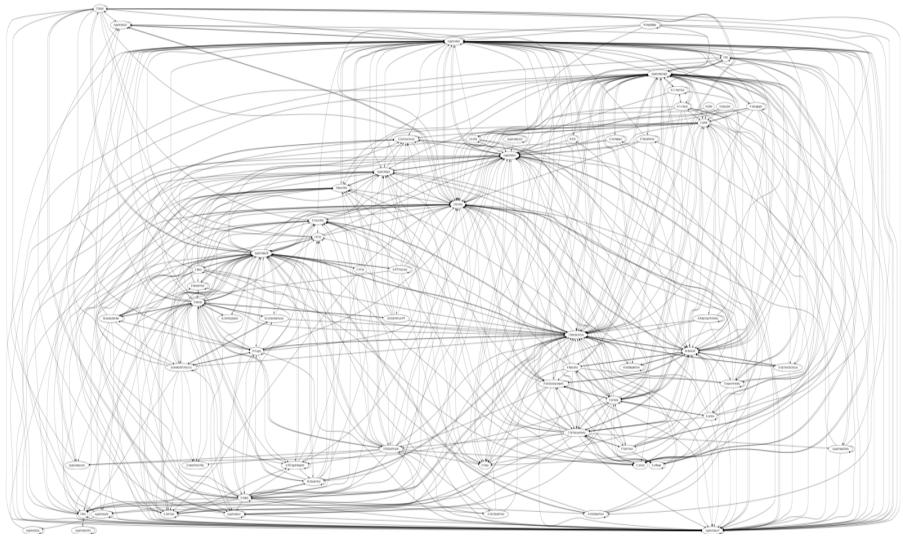


Рис. 15: Компоненты Hadoop, альтернативное размещение элементов.

# Лекция 2: Декомпозиция, объектно-ориентированное проектирование

Юрий Литвинов

yuriilitvinov@gmail.com

## 1. Декомпозиция и модульность

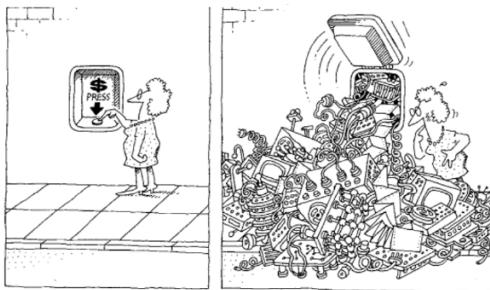
Обсуждение архитектуры программного обеспечения имеет смысл начать с самого начала — что такое сложность и как ею можно управлять. Этот курс больше про объектно-ориентированное программирование, поэтому в этой лекции будет довольно много объектно-ориентированной специфики, но сложность и декомпозиция как способ борьбы со сложностью свойственны всем парадигмам программирования вообще.

### 1.1. Сложность

Архитектура как таковая нужна для разработки сложных программных систем. Интересная особенность программного обеспечения в том, что сложность является неотъемлемой его частью и если попытаться от неё избавиться, разрабатываемое ПО теряет и свою ценность. Это отличает программную инженерию от многих других видов человеческой деятельности — если физики пытаются по сложному природному явлению построить его простую модель, чтобы исследовать её свойства и применить полученные знания обратно к природным явлениям, то у программистов даже модели ПО сложны. Почему так — потому что, как правило, каждая строчка кода уникальна и требует некоторого мыслительного процесса, чтобы её написать; иначе общие части можно было бы вынести или сгенерировать. Так что каждая строчка кода несёт в себе содержательную информацию, которая необходима для работоспособности программы, а поскольку типичное разрабатываемое ПО имеет впечатляющие размеры (см. предыдущую лекцию), то и сложность ПО велика и от неё никуда не деться.

При этом выделяют два “вида” сложности — *существенную сложность* (essential complexity) и *случайную сложность* (accidental complexity). Существенная сложность присуща самой решаемой проблеме, случайная сложность — это сложность, привнесённая способом решения. От случайной сложности можно и нужно избавляться, для этого существует масса полезных приёмов, некоторые из которых будут упомянуты в этой лекции (приёмов, как правило, тактического характера — как спроектировать API, как разделить функциональность по классам, как сделать удобными вызовы методов и т.д.). От существенной сложности избавиться нельзя, она объективно существует в предметной области, а программное обеспечение лишь “моделирует” её в коде. Например, если мы решаем дифференциальное уравнение, вряд ли наш код может быть проще, чем математический метод его решения.

Поэтому деятельность архитектора направлена не на борьбу со сложностью, а на управление ею. И основной приём управления сложностью — это её скрытие. Вот рисунок из замечательной книги G. Booch, “Object-oriented analysis and design”, который иллюстрирует ситуацию:



Система сложна, существенная сложность может быть впечатляющей, но пользователю ничего знать про это не надо, ему система предоставляет возможно более простой интерфейс для решения его задач. Причём под пользователем тут понимается как конечный пользователь системы, так и коллеги-программисты, которые будут пользоваться написанным кодом, вызывать веб-сервисы и т.д.

Интересно, что все сложные системы имеют общие свойства:

- иерархичность — свойство системы состоять из иерархии подсистем или компонентов;
- наличие относительно небольшого количества видов компонентов, экземпляры которых сложно связаны друг с другом;
- сложная система, как правило, является результатом эволюции простой системы.

На самом деле, скорее всего, не все сложные системы обладают этими характеристиками, но ими обладают те системы, о которых мы можем хоть как-то рассуждать. Сложность, которая в принципе не иерархична или не поддаётся разбиению на виды элементов, остаётся “невидимой” для нас, поскольку у нас просто нет механизмов исследования таких систем. Но ежели система всё-таки обладает вышеперечисленными свойствами, то к ней можно применить следующие подходы:

- декомпозиция — разбиение иерархичной системы на компоненты, каждый из которых состоит из более мелких компонентов и т.д., при этом про каждый компонент можно более-менее осмысленно рассуждать в отдельности;
- абстрагирование — выделение общих свойств компонентов, их классификация, рассуждение не о конкретных элементах системы, а об их типах.

Всё это, естественно, вовсю используется в программировании. С декомпозицией и абстрагированием всё и так понятно, а вот то, что сложные системы появляются из простых, влечёт тот печальный факт, что даже изначально небольшие и простые программы желательно проектировать сразу так, будто они превратятся в монстров на много миллионов строк кода.

Ещё одно важное соображение состоит в том, что сложность создаваемых систем вполне может превосходить человеческие возможности. Декомпозиция позволяет рассматривать отдельно каждый компонент или аспект системы, так, чтобы по отдельности в них можно было разобраться, но для реально используемого ПО вполне возможно, что систему целиком не знает никто, и даже никто не в состоянии знать все её детали.

## 1.2. Подходы к декомпозиции

Есть разные способы декомпозиции задачи на подзадачи. Первый — восходящее проектирование, когда сначала создаются отдельные компоненты, а потом из них, как из кирпичиков, собираются более сложные компоненты и, в итоге, система целиком. Такой подход целесообразно применять в исследовательских проектах, когда непонятно, что в конечном итоге может получиться, либо в случае, когда сама предметная область содержит небольшие обособленные задачи, которые можно по-разному комбинировать. Так имеет смысл проектировать библиотеки — рассматривать их как набор кирпичиков, из которых пользователь может сам сложить всё, что ему нужно. Язык программирования FORTH предполагал восходящее проектирование как основной способ создания программ — там программа состояла из *слов*, которые строились из других слов, и т.д. до очень простых конструкций стандартной библиотеки. И, поскольку объявление новых слов было очень простым и их можно было тут же запустить и отладить, рекомендовалось сначала реализовать и отладить простые компоненты системы, а из них постепенно собирать более сложные, тоже запускаясь и отлаживаясь на каждом этапе. В общем-то, современные методологии типа Test Driven Development предполагают похожий подход.

Второй подход — нисходящее проектирование, более “традиционный” в программистском сообществе. Это когда мы сначала рассматриваем задачу целиком, разделяем её на подзадачи, реализуем общую логику, вставляя заглушки вместо реализации подзадач, проверяем, что оно не то чтобы работает, но делает всё, что надо, и в правильной последовательности, потом точно так же рассматриваем каждую отдельную подзадачу, пока не придём к окончательному решению. Такой подход уступает восходящему проектированию в том, что пока мы не дописали систему (или подсистему) до самого низа иерархии декомпозиции, мы не можем её внятно тестировать, зато лучше тем, что направляет процесс проектирования и разработки — меньше шансов реализовать что-то, что потом не понадобится, и вообще, всё более предсказуемо. Такой подход используется, когда есть чёткое видение конечного результата и надо минимальными усилиями этого результата достичь (то есть, на самом деле, почти всегда).

Важным понятием для нисходящего проектирования являются *модули* — структурные единицы кода, которые соответствуют подзадачам, на которые разбита система. Модули в объектно-ориентированных языках могут быть классами или компонентами (иногда целыми отдельными подсистемами, например, веб-сервисом), в функциональных языках — отдельными функциями или какими-либо способами их группировки (в F# они так и называются, *module*), в структурных языках они тоже есть — пара из .h и .c-файлов в C, модули в Паскале, пакеты в Аде и т.д. Модули характеризуются своим *интерфейсом* и *реализацией*. Интерфейсы необходимы модулям в том числе для того, чтобы можно было реализовать заглушки и чтобы упростить дальнейшую интеграцию — перед тем, как заниматься отдельными подзадачами, мы должны чётко зафиксировать интерфейс, по которому будут общаться реализации разных подзадач, и строго ему следовать. Заглушки тоже должны

следовать интерфейсу, но могут иметь пустую или максимально простую реализацию.

Для модулей есть следующие общепринятые правила их проектирования:

- чёткая декомпозиция — каждый модуль занимает своё место в системе, знает, какую задачу он решает, должно быть понятно, как им пользоваться;
- минимизация интерфейса модуля — вечный принцип хорошей архитектуры “меньше знаешь — крепче спиши”; использование модуля должно быть максимально простым, но всё ещё позволять решать задачу;
- один модуль — одна функциональность; если модуль делает что-то и что-то ещё, разбейте этот модуль на два;
- отсутствие побочных эффектов — очень желательное, но не всегда достижимое свойство — чтобы вызов одной и той же функции модуля с одними и теми же параметрами приводил к одним и тем же результатам, и уж тем более не влиял каким-то неочевидным образом на работу других модулей;
- независимость от других модулей — естественно, модуль может использовать другие модули для своей реализации, но он не может знать о деталях реализации других модулей и как-то рассчитывать на эти детали; модуль не имеет права делать предположения о том, кто и как будет его использовать;
- принцип сокрытия данных — внутреннее представление всех данных модуля должно быть известно только ему самому, наружу могут быть видны только абстрактные типы данных, манипулировать которыми можно только через функции модуля.

Модульность как способ разбиения системы на компоненты, вообще говоря, позволяет создавать сколь угодно большие системы, поскольку модуль независим от других и общается с другими только через строго определённые интерфейсы. Разработку каждого модуля можно рассматривать как отдельную задачу, которая меньше по размеру, чем исходная. И проектировщик модуля уровнем выше может вообще ничего не знать про модули уровнем ниже кроме того, как их использовать — что, если соблюдается принцип сокрытия сложности, не должно быть проблемой. Более того, разные модули могут разрабатываться разными людьми или командами, независимо друг от друга, пока все выполняют соглашения, прописанные в интерфейсах (как синтаксические, так и семантические — неправильные типы аргументов поймает компилятор, а вот то, что, например, сначала надо вызвать Connect(), а потом уже GetData() — вряд ли).

При разбиении на модули возникает вопрос, какого размера должен быть модуль, и даже принцип “один модуль — одна функциональность” не всегда даёт ответ на этот вопрос. Начинающие программисты могут пихать вообще всё в один модуль, либо, столь же часто, делать кучу модулей по одной функции в каждом, думая, что вот, красивый модульный дизайн, минимизация интерфейсов и всё такое. Проблема в том, что если модули слишком большие, то сложность каждого модуля неоправданно велика, и, соответственно, неоправданно велики затраты на его разработку. Если модули слишком маленькие, то неоправданно велики затраты на их интеграцию — модулей становится слишком много, взаимодействия между модулями становятся слишком сложны. В вырожденных случаях (один большой модуль или куча модулей по одной функции) мы никаких преимуществ от модульности вообще не получаем. Поэтому должен соблюдаться некоторый баланс:



Есть знаменитое правило “7+2”, говорящее, что человек может одновременно удерживать в голове порядка семи сущностей (плюс-минус, в зависимости от индивидуальных способностей). Это правило может быть хорошей отправной точкой при проектировании системы и разбиении на модули.

Ещё хорошим показателем качества разбиения на модули могут быть численные метрики сопряжения и связности, которые можно просто посчитать (естественно, автоматически) для данного куска кода, или оценить на глаз. *Сопряжение (Coupling)* — мера того, насколько взаимосвязаны разные модули в программе (то есть насколько часто один модуль дёргает другие, насколько много этих других и насколько много они должны знать друг о друге). *Связность (Cohesion)* — мера того, насколько взаимосвязаны функции внутри модуля и насколько похожие задачи они решают. Целью при проектировании является слабое сопряжение (опять-таки, “меньше знаешь — крепче спиши”, можно независимо менять компоненты системы, не боясь всё сломать) и сильная связность (функции внутри модуля должны быть связаны друг с другом, иначе их следует разложить в отдельные модули — прежде всего для того, чтобы модуль было проще понять и использовать).

Отдельно обращаю внимание, что все эти хорошие практики направлены на упрощение понимания системы. И речь в этом курсе идёт не о понимании, которое может получить внимательный читатель, аккуратно разбираясь с каждой строчкой кода, а о понимании, которое необходимо, когда завтра релиз, а где-то в этих сотне тысяч строк кода есть критический баг, без исправления которого релиз не состоится, а код этот писала соседняя команда из Чехии, которая всем составом ушла в отпуск, и все комментарии на чешском.

## 2. Объектно-ориентированное проектирование

### 2.1. Объекты

В объектно-ориентированном проектировании роль модулей играют объекты и классы, а поскольку этот курс ориентирован в основном на объектно-ориентированное программирование, дальше речь пойдёт про них. Впрочем, бывают объектно-ориентированные языки, не имеющие вовсе понятия “класс”, так что поговорим сначала именно про объекты. Вот несколько разных определений из нескольких разных источников:

- Objects may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods — [Wikipedia](#)
- An object stores its state in fields and exposes its behavior through methods — [Oracle](#)

- Each object looks quite a bit like a little computer — it has a state, and it has operations that you can ask it to perform — [Thinking in Java](#)
- An object is some memory that holds a value of some type — [The C++ Programming Language](#)
- An object is the equivalent of the quanta from which the universe is constructed — [Object Thinking](#)

The C++ Programming Language подходит к определению объекта наиболее прагматично, потому что, вообще говоря, там это понятие нужно для описания семантики языка, а не для философских рассуждений, но вместе с тем такое определение самое бесполезное, поскольку перекладывает всю ответственность на понятие “type”. Определения из Википедии и из доков по Java слишком механистичны и следуют традиции ошибочного понимания объектов как структур с методами — технически это так, но не в этом суть понятия “объект”. Больше всего соответствует этому курсу определение из Thinking in Java, объект как изолированная сущность, которая обладает каким-то поведением и может иметь состояние (что отличает объектно-ориентированное программирование от функционального). Object Thinking даёт слишком философское, хотя и совершенно в духе этого курса определение — объект как единица декомпозиции объектно-ориентированной программы.

В любом случае, объекты имеют три важных свойства: состояние, поведение и идентичность. С состоянием связано такое важное понятие, как *инвариант* — набор логических условий, которые должны исполняться всё время жизни объекта. Инвариант важен, поскольку позволяет реализовывать методы будучи уверенными в том, что эти условия выполнены — например, что количество элементов в связном списке равно значению поля `length`, что позволяет во всех методах, требующих просмотра всего списка, бежать просто от 0 до `length` и не думать, что будет, если следующий указатель `null`. Каждый объект сам отвечает за поддержание своего инварианта и обязан не давать возможности нарушить его извне (поэтому public- поля запрещены, например).

Поведение объекта принципиально отличается от вызова функции модуля тем, что объект вправе сам решать, как обработать пришедший к нему запрос, так что правильнее всего считать, что объекты не вызывают методы друг друга, а отправляют друг другу сообщения. Более конкретно, методы объекта могут быть полиморфными, так что код, который будет реально вызван, неизвестен вызывающему (и может даже не существовать на момент, когда вызывающий написан и скомпилирован). То, какой код будет исполнен при полиморфном вызове, определяется типом времени выполнения сразу двух разных объектов, это называется “двойная диспетчеризация” и подробности про это будут, когда дойдём до паттерна “Посетитель”. Объекты также могут существовать в разных потоках и обрабатывать вызовы асинхронно, теоретически это можно сделать прозрачно для вызывающего (есть архитектуры, где каждому объекту всегда сопоставляется отдельный поток, например, система RTST Андрея Николаевича Терехова).

Класс — это тип объекта. Классы определяют структуру данных, которые хранят объект, и методы (с их реализацией, обратите внимание), которые есть у каждого объекта этого класса. Класс — это сущность времени компиляции (и именно с классами в основном работают программисты), объект — сущность времени выполнения (с ними программисты

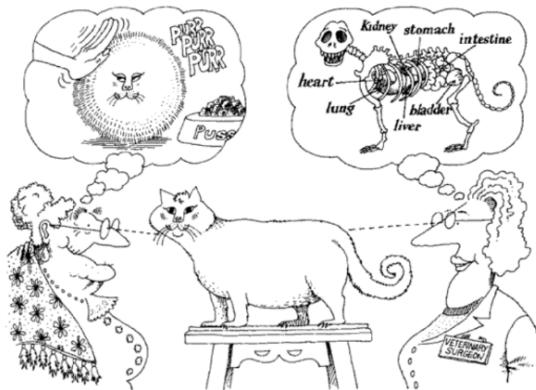
работают, например, отлаживая систему). Но, как уже упоминалось выше, есть языки, не имеющие понятия “класс” (например, JavaScript до версии ES6, Smalltalk).

## 2.2. “Три кита” объектно-ориентированного программирования

### 2.2.1. Абстракция

Есть три основных принципа объектно-ориентированного программирования, про которые обычно рассказывают на первом курсе и пишут в любой книге — инкапсуляция, наследование, полиморфизм. Обсудим некоторые из них ещё раз, уже с архитектурной точки зрения. Но начнём мы с абстракции — на самом деле, самого важного понятия ООП, хотя она и не относится к “китам”. Потому что она касается не только ООП, но и всего остального — структурного, функционального программирования, алгебры и вообще способности людей к мышлению.

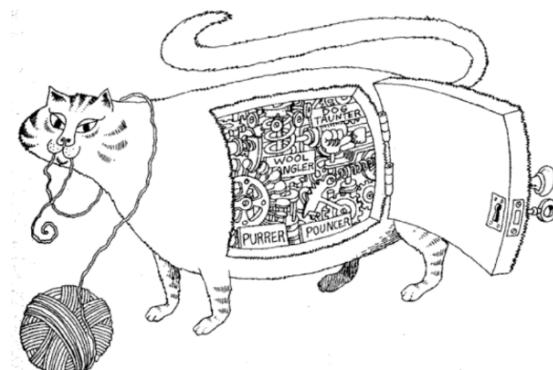
Абстракция выделяет существенные характеристики объекта, отличающие его от остальных объектов, с точки зрения наблюдателя. Один и тот же физический объект может обладать несколькими разными абстракциями одновременно, как на ещё одной картинке из книги Буча “Object-oriented analysis and design”:



Наличие хорошей и аккуратно спроектированной абстракции в разы упрощает работу с системой. Абстракция может иметь несколько взаимозаменяемых реализаций, и если абстракция достаточно содержательна, на ней могут базироваться реализации других абстракций. Хороший пример использования абстракций — вся математика. Например, в алгебре вводятся алгебраические структуры, такие как кольцо, поле, моноид — абстракции, декларирующие наличие определённых свойств у соответствующих этим абстракциям объектов. Для алгебраических структур доказываются теоремы, которые ничего не знают про реализацию структуры, а знают только то, что декларируется абстракцией (например, для любого моноида верно то-то). Реализации могут быть самые разные — множество целых чисел и операция сложения образуют моноид, строки и операция конкатенации тоже его образуют, функции из  $\text{int}$  в  $\text{int}$  и операция композиции — тоже моноид. Значит, для них верны все теоремы, которые доказаны для моноидов. Теорема — это внешний код, пользующийся абстракцией, моноид — абстракция, композиция функций — реализация абстракции.

## 2.2.2. Инкапсуляция

Инкапсуляция на самом деле разделяет интерфейс (*контракты*) абстракции и её реализацию. Инкапсуляция опять-таки реализует принцип “меньше знаешь — крепче спиши”, позволяя пользователю не знать про реализацию вообще. Но главное, что инкапсуляция защищает инварианты абстракции от их порчи извне. И ещё одна картинка из книжки Буча по этому поводу:



## 2.2.3. Наследование

*Наследование* часто определяют как то, что потомок получает все `public`- и `protected`- поля и методы предка. Это неправильно (то есть правильно технически, обычно, но неправильно по сути). Наследование — это отношение “является” (*is-a*) между типами, ну или более формально, один из способов реализации *сабтайпинга* в системе типов. Правильнее всего понимать наследование как “Объект типа-потомка является одновременно объектом типа-предка, поэтому может использоваться везде, где может использоваться предок”. Это важно понимать, чтобы избегать семантических нарушений иерархии наследования (“транзационное” определение наследования этому только мешает).

Простой пример на понимание наследования — это загадка, которая иногда всё ещё упоминается на собеседованиях — что от чего надо наследовать, прямоугольник от квадрата или квадрат от прямоугольника? Конечно квадрат от прямоугольника, не задумываясь отвечает собеседуемый, ведь квадрат — это такой прямоугольник, у которого все стороны равны. Можно и прямоугольник унаследовать от квадрата, в конце концов и тот и другой имеют длину и ширину, у них можно посчитать площадь, с `public`- и `protected`- полями и методами всё ок. Но если мы напишем код, который увеличивает площадь прямоугольника вдвое, удвоив его длину, то для квадрата этот код работать уже не будет — у квадрата есть инвариант, которого нет у прямоугольника — длина всегда равна ширине. Так что квадрат не может использоваться везде, где может использоваться прямоугольник, и наследовать его от прямоугольника нельзя. Но и наоборот нельзя, опять-таки из-за инварианта — код, увеличивающий сторону квадрата, может сломаться, получив прямоугольник. Итого, внезапно, прямоугольник и квадрат вообще не должны быть связаны наследованием. Важная мораль этой истории — потомок наследует все инварианты предка и не вправе их нарушать (даже если это такие неявные инварианты, как “при увеличении одной из сторон вдвое площадь увеличится вдвое”).

Всему этому противоречит наличие `private`-наследования в C++. Дело в том, что во времена C++ наследование рассматривалось как способ переиспользования кода — выносим общий код в предка, наследуемся несколькими потомками, получаем в каждом этот общий код. Современная архитектурная мысль рекомендует так не делать (и правда, `private`-наследование в настоящем коде на C++ встречается крайне редко). Наследование применяется только с двумя целями — как средство классификации и абстрагирования в соответствии с уже существующими системами классификации в предметной области и как средство обеспечения полиморфизма. И для того, и для другого случая достаточно наследования интерфейсов, так что наследование конкретных классов друг от друга нынче большая редкость, а абстрактные классы применяются с сугубо pragматическими целями — чтобы лишний раз поля не писать (и тоже редко).

Вместо наследования для переиспользования кода нынче можно использовать *композицию* — отношение “имеет” (`has-a`) между типами, которое реализуется обычно с помощью `private`-полей. Общая функциональность выносится в отдельный класс и все, кому она нужна, просто включают объект этого класса себе как поле (если повезёт, класс окажется статическим, тогда даже поле не нужно). Композиция позволяет создавать, менять и удалять связи прямо во время выполнения, а также делегировать запросы к объекту-“хозяину” объекту, который внутри, что позволяет делать всякие хорошие вещи типа паттернов “Декоратор”, “Прокси”, “Адаптер” и т.д., про которые будет дальше в этом курсе.

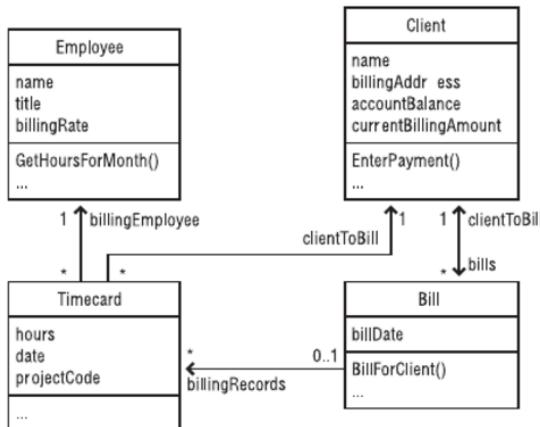
Интересно, что наследование и композиция более-менее взаимозаменяемы. Технически, наследование и есть композиция — объект-потомок всегда включает в себя объект класса-предка. Можно переделать одно в другое, особенно если язык поддерживает множественное наследование, правда, для этого часто приходится довольно радикально менять точку зрения на систему. Композиция, как уже говорилось, считается более предпочтительной, чем наследование, поскольку наследование фиксировано во время компиляции, а композицию можно менять во время выполнения, что даёт большую гибкость.

## 2.3. Выделение абстракций

Теперь надо обсудить, откуда, собственно, брать объекты и классы, если система ещё не написана. Допустим, у нас есть техническое задание, требующее разработать приложение для какой-то (возможно, незнакомой нам) предметной области. Тогда нашей первой задачей будет построение объектной модели предметной области и решаемой задачи, для чего обычно выполняются следующие действия:

1. определение объектов и их атрибутов, чаще всего на основе общения с экспертами, самостоятельного изучения предметной области и здравого смысла — часто встречающиеся существительные в речи экспертов, скорее всего, станут классами;
2. определение действий, которые могут быть выполнены над каждым объектом (значение ответственности) — так же, слушаем экспертов и собираем информацию о предметной области, действия — это глаголы;
3. определение связей между объектами, задавая вопросы в духе “а про это кто знает”, “а это где используется” и т.д.
4. определение интерфейса каждого объекта, когда фиксируется и формализуется всё, что удалось сделать на предыдущих этапах.

Результатом этой деятельности будет первый грубый набросок архитектуры системы, выражаемый обычно в виде диаграммы классов:



Классы, конечно, появляются не только из предметной области, но и могут появляться в процессе реализации. Часто встречающиеся источники абстракций таковы:

- изоляция сложности — имеет смысл делать отдельными классами сложные алгоритмы, нетривиальные структуры данных, целые сложные подсистемы прятать за простыми фасадами. Делается это для того, чтобы, во-первых, спрятать сложность от остальной системы, во-вторых, иметь возможность менять алгоритмы, оптимизировать внутреннее представление и т.д., зная, что это ничего не сломает. При этом надо тщательно проектировать интерфейсы абстракций, скрывающих сложность, потому что они имеют тенденцию сами становиться слишком сложными, что убивает все преимущества инкапсуляции.
- Изоляция возможных изменений — прятать всё, что имеет большие шансы поменяться, в отдельный класс или подсистему, чтобы их можно было выкинуть и переписать, если изменение таки произойдёт. Однако не надо переусердствовать, хорошая архитектура позволяет себя менять и без специальной поддержки изменчивости. Основные источники изменений:
  - бизнес-правила, то есть алгоритмы из предметной области, которые реализует программа, они меняются на удивление часто;
  - зависимости от оборудования и операционной системы — меняются редко, но изменение может быть критично для жизни программного продукта — ваше любимое железо через пять лет, скорее всего, никто не будет выпускать, что при неаккуратном проектировании может потребовать переписать всю систему с нуля. Если вы думаете “кому мой программный продукт будет нужен через пять лет”, вспомните (или почитайте) про проблему 2000 года;
  - ввод-вывод, как формат файлов сохранения, так и формат сетевых пакетов, набор допустимых команд и т.д., они наверняка будут почему-то меняться;

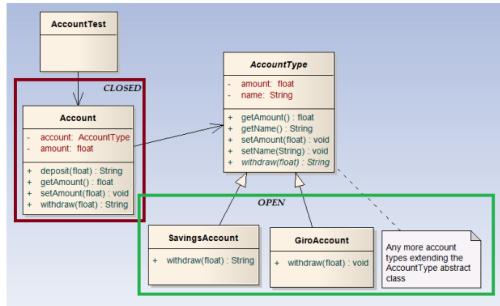
- нестандартные возможности языка — система должна иметь возможность пережить смену компилятора;
  - сложные аспекты проектирования и конструирования — это обсуждалось выше, сложность полезно прятать, но не только сложность реализации, но и сложность архитектуры — сложные решения часто оказываются неверными, их должно быть можно легко поменять;
  - третьесторонние компоненты — нельзя критически зависеть от чего-то, что перестанет поддерживаться через полгода или станет стоить бешеных денег.
- Изоляция служебной функциональности — сущности, которые нужны для работы других сущностей. Это различного рода фабрики, репозитории, диспетчеры, медиаторы, сервисы (группы статических методов, полезных для работы) и т.д. и т.п. Классы, отвечающие за бизнес-логику, должны содержать только бизнес-логику, не захламляя её деталями реализации, детали должны жить отдельно.

## 2.4. Принципы SOLID

Есть пять базовых принципов объектно-ориентированного проектирования, известные как принципы SOLID. Эти принципы должны применяться при проектировании всех объектно-ориентированных систем, их любят спрашивать на собеседованиях, и вообще, их следовало бы рассказывать ещё на первом курсе, но обычно этого не делают.

Принципы таковы:

- Single responsibility principle, принцип единственности ответственности — каждый класс должен делать что-то одно. Кажется привлекательным иметь “швейцарский нож”, который бы один решал все возможные проблемы, но такой класс, во-первых, тяжёл в сопровождении, а во-вторых, сложно понять его роль в системе, сложно объяснить её новым людям в проекте. Поэтому про каждый класс должно быть можно в одном предложении сказать, зачем он нужен, например, “утилиты для работы со строками”, “вычислялка процентов по вкладу”. И поэтому важно писать комментарии к самому классу — если у вас не получается сформулировать кратко, что делает этот класс, значит, это плохая абстракция и её надо разбить на несколько классов. Ещё надо следить, чтобы ответственность класса была полностью инкапсулирована в этом классе. То есть, например, если у вас есть “утилиты для работы со строками” и “ещё утилиты для работы со строками”, дела ваши плохи — вы никогда в жизни не запомните где нужный вам метод. Это всё касается не только классов, но и функций, и целых подсистем.
- Open/closed principle, принцип открытости/закрытости — абстракция (класс, модуль, функция) должна быть открыта для расширения, но закрыта для изменения. То есть, если интерфейс уже стабилизировался и им начали пользоваться, менять его нельзя. Если надо добавить в абстракцию новую функциональность, можно использовать наследование и/или заранее подготовленные точки расширения, как на картинке:

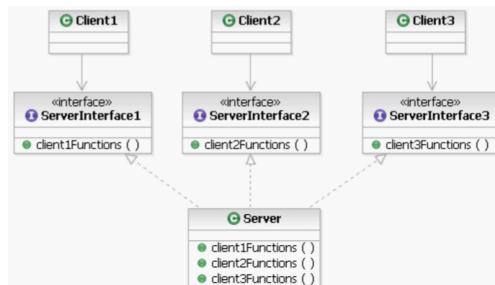


Если это правило не соблюдать, интерфейс абстракции начнёт увеличиваться в размерах, сам собой начнёт нарушаться принцип единственности ответственности, и в результате мы получим несопровождаемого и неизабельного монстра (см. анти-паттерны “God object” и “Swiss army knife” далее в этом курсе).

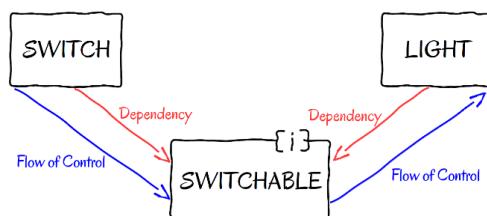
- Liskov substitution principle, принцип подстановки Барбары Лисков — то самое определение наследования, о котором шла речь выше. Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом. Это означает, в частности, что классы-потомки должны реализовывать интерфейсы классов предков (как чисто синтаксически, так и семантически — делать то, что ожидается от предка) и, про что часто забывают, выполнять все инварианты классов-предков, явные или неявные. Хороший пример проблем с принципом подстановки — проверяемые исключения в Java. Методы класса-предка могли задекларировать исключения, которые они могут бросать, тогда потомок, переопределяющий эти методы, имел право бросать только эти исключения. Почему — вызывающий мог использовать try/catch для обработки ошибок, и если там были написаны catch-блоки для исключений предка, а потомок бросал что-то новое, это ломало вызывающего и тем самым нарушило принцип подстановки. Компилятор проверял такие ситуации и сообщал об ошибке, если потомок пытался бросить незадекларированное в предке проверяемое исключение. Проблема в том, что в большинстве случаев заранее предсказать все исключительные ситуации, которые могут возникнуть в потомках, при написании предка невозможно, поэтому проверяемые исключения особо не используются нынче в Java и так и не попали ни в один другой язык.



- Interface segregation principle, принцип разделения интерфейсов — клиенты не должны зависеть от методов, которые они не используют. Если у абстракции есть группы методов, нужные разным клиентам, то имеет смысл сделать разные интерфейсы, по одному на каждую группу методов. Например, у нас есть модем (например, 4G/LTE), который умеет устанавливать соединение и при наличии соединения передавать данные. У нас есть системная утилита установки соединения, которая следит за сотовой сетью и устанавливает соединение, если это возможно, и у нас есть сетевой стек, который передаёт данные, если его попросят. Если просто сделать интерфейс IModem, который будет иметь методы Connect() и Send(), это как раз нарушит принцип разделения интерфейсов, потому что утилиту установки соединения надо будет пересобирать каждый раз, когда меняются параметры метода Send(), который ей совсем не нужен, а сетевой стек придётся пересобирать каждый раз, когда меняется Connect(), который в этом примере сетевой стек не использует (например, потому, что не хочет знать подробности сотовой связи). Необходимость разделять интерфейсы абстракции, впрочем, может указывать на нарушение принципа единственности ответственности, но может и нет (в нашем примере “модем” был единой сущностью и в этом плане всё было ок, просто клиенты использовали его по-разному). Ещё важно не переусердствовать, ведь формально у класса должно быть ровно столько интерфейсов, сколько клиентов его используют, но это бессмысленно и опасно в плане излишних зависимостей. Группировать в интерфейсы надо методы, которые реально кластеризуются по смыслу, а не чисто механически.



- Dependency inversion principle, принцип инверсии зависимостей — модули верхних уровней не должны зависеть от модулей нижних уровней, оба типа модулей должны зависеть от абстракций. Этую фразу, бывает, студенты выучивают наизусть и повторяют на экзамене как заклинание, совершенно не понимая её смысла. Проще всего пояснить суть дела на примере:



Положим, у нас есть выключатель, который управляет лампочкой. Наивным решением было бы сделать класс `Switch`, у которого было бы поле типа `Light` (лампочка), и он бы вызывал её методы `on()`/`off()`. Чем это плохо — выключатель может управлять только лампочкой, выключатель надо перекомпилировать, когда меняется лампочка, работу выключателя невозможно понять, не зная про лампочку. Если выключатель ещё и сам создаёт лампочку, невозможно подсунуть вместо неё мокс-объект, что затрудняет тестирование. Правильное решение (та самая инверсия зависимостей) — сделать интерфейс “то, что можно включать/выключать” (`Switchable`), сделать так, чтобы лампочка реализовывала этот интерфейс и сделать в классе `Switch` поле типа `Switchable`, так, чтобы он вообще ничего про лампочку не знал. Тперерь выключатель сам объект-лампочку создать в принципе не может (он просто не знает, что она есть), так что должен получать свою зависимость либо параметром в конструктор, либо в метод-сеттер (что называется `Dependency Injection`, внедрение зависимости). Кстати, существуют и активно используются целые библиотеки, занимающиеся управлением зависимостями и инициализацией объектов, так называемые `IoC`-контейнеры (`Inversion of Control`), без них не обходится ни одна современная библиотека для разработки веб-приложений, например. Тем не менее, и без библиотек это хорошая практика, существенно уменьшающая зависимости между модулями программы.

## 2.5. Закон Деметры

Ещё одно хорошее правило, не являющееся частью принципов SOLID — это закон Деметры, кратко формулирующийся как “Не разговаривай с незнакомцами”. Более формально, правило говорит о том, что объект А не должен иметь возможность получить непосредственный доступ к объекту С, если у объекта А есть доступ к объекту В, и у объекта В есть доступ к объекту С. Например,

```
book.pages.last.text // Плохо  
book.pages().last().text() // Лучше, но тоже не супер  
book.lastPageText() // Идеально
```

Почему так — вызовы “по цепочке” раскрывают внутреннюю структуру данных класса и не дают её потом изменить. В нашем примере с книгой цепочка вызовов говорит, что в книге есть коллекция страниц, у которой обязательно есть метод `last()` (и любая другая коллекция без этого метода не подойдёт), и что там лежат страницы, у которых есть метод `text()`. Если мы хотим просто получить текст последней страницы, это ну очень много знаний, которые нам не нужны и заставляют нас зависеть от автора класса-книги. В книжке Р. Мартина “Чистый код” проводится аналогия с крушением поезда — где-то посреди структуры меняется внутреннее представление и всё, все эти цепочки вызовов надо переделывать.

Как обычно, главное не переусердствовать. Код вида `book.firstPageText()`, `book.secondPageText()`, `book.thirddPageText()` и т.д. гораздо хуже, чем “паровозик”. Закон Деметры говорит просто, что не надо выставлять напоказ структуру данных, которая не является частью абстракции, и если мы реально ожидаем доступ к каждой странице (что, вообще говоря, для книги вполне ок), то имеет смысл предоставить-таки коллекцию `pages()`. Более того, с нарушением закона Деметры часто путают очень полезный приём

программирования “Fluent API”, когда через точку пишется последовательность действий, которые надо выполнить над объектом (хороший пример — это Java Stream API или LINQ в .NET), этот же приём используется в паттерне “Строитель”, про который тоже будет в этом курсе. Здесь с законом Деметры всё ок, потому что цепочка fluent-вызовов, как правило, возвращает один и тот же объект, не раскрывая его внутренней структуры (любители функционального программирования могут провести параллели с монадами, и на самом деле LINQ проектировался как самые настоящие монады на C#).

## 3. Некоторые принципы хорошего кода

Теперь перейдём от абстрактных рассуждений к конкретным примерам кода и соображениям по поводу “тактического” проектирования, помогающим писать расширяемый и сопровождаемый код. Далее будет на самом деле краткий обзор книжек Р. Мартина “Чистый код” и “Code Complete” С. Макконелла, которые, впрочем, рекомендуются к прочтению — они не очень большие, легко читаются и могут научить куче полезных приёмов.

### 3.1. Абстракция

Начнём мы с абстракции, как самого важного из “китов” ООП, хоть её редко упоминают в таковом качестве.

Рассмотрим пример. У нас есть объект “Шрифт” и мы хотим уметь менять ему размер. Вот так это можно было бы сделать:

- `currentFont.size = 16` — прямое присвоение полю, ужасно. Во-первых, шрифт не может поддерживать свой инвариант и изменить внутреннее представление размера. Во-вторых, тут написано “размеру присвоить 16”. Окей, чего 16? Ну а чего размеру?
- `currentFont.size = PointsToPixels(12)` — чуть лучше. Теперь размер, видимо, изменяется в пикселях.
- `currentFont.sizeInPixels = PointsToPixels(12)` — ещё чуть лучше. Теперь про размер явно написано, что он измеряется в пикселях.
- `currentFont.setSizeInPoints(sizeInPoints)`  
`currentFont.setSizeInPixels(sizeInPixels)` — совсем хорошо. Нам-то какое дело, в чём внутри у шрифта измеряется размер, мы хотим его выставлять в тех единицах измерения, которые нам удобны.

Ещё один небольшой пример, обычного класса на Java, который умеет, видимо, что-то выполнять и печатать по этому делу отчёты:

```
public class Program {  
    public void initializeCommandStack() { ... }  
    public void pushCommand(Command command) { ... }  
    public Command popCommand() { ... }  
    public void shutdownCommandStack() { ... }  
    public void initializeReportFormatting() { ... }  
}
```

```
public void formatReport(Report report) { ... }
public void printReport(Report report) { ... }
public void initializeGlobalData() { ... }
public void shutdownGlobalData() { ... }
}
```

Тут никаких полей нет, имена методов и параметров адекватны, но всё равно этот класс ужасен. Потому что нарушает принцип единственности ответственности. Он умеет и что-то делать со стеком команд, и с отчётом, и с глобальными данными, так что непонятно, когда и как им пользоваться. Про этот класс нельзя одним предложением сказать, что он такое. Чтобы сделать этот класс хорошим, его неплохо бы распилить на три — стек команд, управлялка отчётом и хранилище глобальных данных. И добавить четвёртый класс, который бы управлял этими тремя классами. И это не будет овердизайном, потому что тут и так на самом деле три класса, просто они упиханы в один.

Вот другой класс на Java, представляющий сотрудника какой-то компании:

```
public class Employee {
    public Employee(
        FullName name,
        String address,
        String workPhone,
        String homePhone,
        TaxId taxIdNumber,
        JobClassification jobClass
    ) { ... }

    public FullName getName() { ... }
    public String getAddress() { ... }
    public String getWorkPhone() { ... }
    public String getHomePhone() { ... }
    public TaxId getTaxIdNumber() { ... }
    public JobClassification getJobClassification() { ... }
}
```

У этого класса с абстракцией всё хорошо, он занимается ровно одним делом (представляет сотрудника компании), все имена понятны. Кроме того, тут используется ещё один хороший для абстракции приём — специальные типы для значений. Например, TaxId вполне может быть классом с ровно одним полем типа String, геттером и сеттером, и всё. Зато теперь тут скрыто внутреннее представление TaxId (и его легко можно переделать на int, если окажется, что это возможно) и компилятор поругается, если мы перепутаем порядок аргументов. FullName может быть несколько более сложной штукой, там может быть фамилия, имя и отчество. Мы это всё тоже могли бы передавать в конструктор Employee как три отдельные строки, но зачем, ведь FullName — это концептуально целостная штука. Достоинства такого подхода понятны, недостатки тоже — за абстракцию и большую надёжность мы платим строками кода. Стоит ли платить эту цену — приходится решать в каждом конкретном случае. Отметим, что в функциональных языках объявить тип обычно

гораздо менее трудозатратно, чем в Java, так что там таким приёмом пользуются часто и с удовольствием.

В продолжение темы платы строками кода за качество абстракции ещё один пример:

```
public class Point {  
    public double x;  
    public double y;  
}
```

против

```
public interface Point {  
    double getX();  
    double getY();  
    void setCartesian(double x, double y);  
    double getR();  
    double getTheta();  
    void setPolar(double r, double theta);  
}
```

Тут мы видим, что поскольку в Java нет структур, то приходится объявлять классы, и чтобы не писать ненужных геттеров/сеттеров, можно сразу объявить public- поля, потому что по смыслу это структура, инвариантов у неё нет, и использовать её предполагается только как хранилище данных. Тем не менее, несмотря на большую многословность, второй вариант более архитектурно правильный, потому что обладает интересным свойством — тут вообще нигде не написано, как точка представляется внутри. Мы можем трактовать точку как точку в декартовых координатах, можем — как в полярных, ей всё равно. Для этого, кстати, геттеры позволяют получить каждую координату отдельно, но сеттеры позволяют выставить только обе координаты сразу — позволять менять координаты по одной было бы странно с точки зрения семантики операций с учётом того, что изначально точка могла задаваться в другой системе координат. Кстати, этот пример показателен не только для Java, где всё очень плохо, но и для C#, где есть и структуры и свойства — первая абстракция всё равно отличается от второй тем, что она “недостаточно абстрактна”. Нужна ли нам большая абстрактность или это приведёт к овердизайну — вопрос, на который должен ответить архитектор, опять-таки, для каждого конкретного случая.

Следующее соображение, влияющее на качество абстракции — это её “уровень” и консистентность этого уровня по абстракции. Например:

```
public class EmployeeRoster implements MyList<Employee> {  
    public void addEmployee(Employee employee) { ... }  
    public void removeEmployee(Employee employee) { ... }  
    public Employee nextItemInList() { ... }  
    public Employee firstItem() { ... }  
    public Employee lastItem() { ... }  
}
```

Тут принцип единственности ответственности соблюдается — это список сотрудников какой-то компании, всё хорошо. Наследование используется по делу, список сотрудников

определенно является списком. Имена и параметры тоже адекватны — можно добавить сотрудника, получить сотрудника, удалить сотрудника, вроде всё ок. Тем не менее, сравним это со вторым классом:

```
public class EmployeeRoster {  
    public void addEmployee(Employee employee) { ... }  
    public void removeEmployee(Employee employee) { ... }  
    public Employee nextEmployee() { ... }  
    public Employee firstEmployee() { ... }  
    public Employee lastEmployee() { ... }  
}
```

Тут мы уже не наследуемся от списка (о ужас, теперь для нашего класса не будут работать библиотечные алгоритмы, работающие над списками) и переименовали пару методов. Стало лучше? Гм, рассмотрим ещё один пример — в стандартной библиотеке Java долгое время стек наследовался от списка. Ну а что, стек — это список с методами push и pop. Теперь это каноничный пример плохого дизайна — стек, конечно, не список, потому что не должен получать от списка пачку методов, ломающих инварианты стека. Но суть даже не в инвариантах, а в том, что стек — это что-то простое, список — это что-то сложное и предназначено не для того. Возвращаясь к нашему примеру, список сотрудников — это что-то сложное и техническое, у него есть методы для итерирования (`nextItemInList`, `firstItem` и `lastItem`) и мало ли что ещё. А мы хотим просто список сотрудников, мы не хотим итератор или какой-нибудь сплиттератор, если речь идёт о Java (может, мы двенадцатилетний сын директора компании, который пишет на Питоне автоматизацию начисления зарплаты). Поэтому хорошая абстракция часто вынуждена не расширять уже существующую, а прятать её и предоставлять более простой интерфейс, даже несмотря на то, что это усложнит переиспользование кода. Задача абстракции — быть удобной для использования и для понимания, а не уметь делать много всего. При этом удобство понимания важнее.

Есть некоторый набор общих рекомендаций, про которые тоже неплохо бы помнить, проектируя свои классы и интерфейсы.

- Про каждый класс знайте, реализацией какой абстракции он является. Принцип единственности ответственности имеет очень простой критерий — если про класс можно одним коротким предложением сказать, что он такое, то всё ок. Может потребоваться разделить класс на несколько разных классов просто потому, что методы по смыслу слабо связаны, и это будет не овердизайн, как мы видели ещё в первом примере.
- Учитывайте противоположные методы (`add/remove`, `on/off`, ...). Даже если они сейчас не нужны, когда-то кому-то неизбежно понадобятся.
- Разделяйте команды и запросы, избегайте побочных эффектов. Команда только меняет состояние, не возвращая результат, запрос только возвращает результат, не меняя состояния. Это хорошо как для понимания программы, так и по более практическому соображению — в многопоточной программе синхронизации требуют только команды. Сколько угодно запросов может исполняться параллельно.

- Не возвращайте null. null всегда сюрприз для вызывающего, даже если он прекрасно знает, что любой ссылочный тип может иметь значение null. Возвращайте Option — это вынудит вызывающего явно проверить наличие значения. Ну или бросайте исключение, если null-ы бывают только если что-то пошло не так.
- По возможности делайте некорректные состояния невыразимыми в системе типов. Это особо хорошо работает в функциональных языках, потому что там мощные системы типов, но и в других языках систему типов можно заставить работать на себя — вспомните второй пример, с TaxId и FullName у Employee.
- Семантику языка тоже надо заставить работать на себя в плане чистоты абстракций. Самый простой пример — комментарии в духе “не пользуйтесь объектом, не вызвав init()” можно заменить конструктором. Вообще, всё, что не проверяется компилятором, будет использовано неправильно, так что надо стараться так, чтобы вашей абстракцией пользоваться неправильно было просто невозможно.
- При рефакторинге надо следить, чтобы интерфейсы не деградировали. Рефакторинг — опасная вещь, потому что в погоне за тактической выгодой типа скорости работы или удобства вызовов можно потерять стратегические преимущества хороших абстракций.

## 3.2. Инкапсуляция

Следующее важное понятие в ООП — это инкапсуляция. Под которым обычно понимают сокрытие деталей реализации, несмотря на то, что инкапсуляция — это обычно просто свойство кода, относящегося к одной задаче, лежать рядом. В принципе, в ООП и инкапсуляция в этом понимании, и сокрытие деталей реализации обеспечиваются классами (либо чем-то похожим), так что всё вместе называть инкапсуляцией вполне валидно.

Инкапсуляция — это механизм защиты инвариантов объекта на самом деле. Из этого следует, во-первых, принцип минимизации доступности методов, который коротко формулируется как “меньше знаешь — крепче спиши”. Чем меньше интерфейс объекта, тем проще им пользоваться и тем проще уследить за инвариантами объекта. С другой стороны, тем меньше объект умеет делать, но это даже хорошо — принцип единственности ответственности и всё такое. Другое дело, что если у всех объектов ровно по одному методу (как в функциональных программах, например), то это не идеальная инкапсуляция, а наоборот печаль, потому что сложность переносится из объектов во взаимодействие между объектами.

Во-вторых, из этого следует, что паблик-полей не бывает. Паблик- поля вообще не могут поддерживать никаких инвариантов, поэтому, скорее всего, просто не относятся к тому классу, в котором объявлены. Это, конечно, вызывает вопросы “А что делать, если класс не имеет инвариантов и так, например, используется только для передачи данных?”. Например,

```
class Point {
    public float x;
    public float y;
    public float z;
}
```

и

```
class Point {  
    private float x;  
    private float y;  
    private float z;  
    public float getX() { ... }  
    public float getY() { ... }  
    public float getZ() { ... }  
    public void setX(float x) { ... }  
    public void setY(float y) { ... }  
    public void setZ(float z) { ... }  
}
```

По этому поводу бывают разные мнения, во многих языках есть структуры специально для таких дел, но в целом сообщество сходится на том, что второй вариант предпочтительнее, несмотря на то, что его стоимость в строках кода в разы выше. Почему — код, к несчастью, имеет свойство развиваться, так что то, что раньше было просто классом для передачи данных без инвариантов или чего бы то ни было, внезапно становится полноценной абстракцией с кучей методов, инвариантов и т.д. и т.п. Геттеры с сеттерами легко подправить так, чтобы они начали делать что-то умное, поля — придётся переписывать весь код, использующий наш класс.

Ещё некоторые рекомендации касательно инкапсуляции.

- Класс не должен ничего знать о своих клиентах. Образ мыслей “а здесь я не буду вставлять проверку на null, потому что из моего кода его никто с null не вызывает” неправильный, это нарушение инкапсуляции “с другой стороны” (когда не внешний мир лезет в класс, а класс чего-то хочет от внешнего мира). Всё сломается, когда ваш класс захотят переиспользовать где-то ещё.
- Лёгкость чтения кода важнее, чем удобство его написания. Написать код надо один раз (при этом IDE будет помогать всячими автодополнениями), а читать его придётся, быть может, тысячи раз разным людям.
- Опасайтесь семантических нарушений инкапсуляции, например, “не будем вызывать ConnectToDB(), потому что GetRow() сам его вызовет, если соединение не установлено”. Даже если это правда, это предположение о том, как реализована абстракция, или программирование сквозь интерфейс.
- Protected- и package- полей тоже не бывает. На самом деле, у класса два интерфейса — для внешних объектов и для потомков (может быть отдельно третий, для классов внутри пакета, но это может быть плохо). Все эти интерфейсы должны быть полноценными интерфейсами, то есть защищать инварианты, поддерживать контракты и т.д. Потомков вы вообще не контролируете и доверять им нельзя, классы внутри пакета безопаснее, но тоже никто не сказал, что ваш пакет будет всегда разрабатывать только вы и только в непомутнённом состоянии рассудка.

### 3.3. Наследование и полиморфизм

Наследование хоть и считается одним из трёх китов, на которых зиждется ООП, современная архитектурная мысль считает его очень нишевым и довольно неуклюжим инструментом. Наследование почти всегда может быть заменено на агрегацию с делегированием запросов агрегируемому объекту, и на самом деле внутри в большинстве языков программирования наследование реализуется именно так — класс-потомок просто имеет внутри себя объект класса-предка, методы которого можно вызывать по интерфейсу потомка. Агрегация или композиция имеют ряд преимуществ над наследованием, основное из которых — возможность переконфигурирования во время выполнения. Наследование фиксируется во время компиляции и требуется перекомпилировать программу, чтобы поменять предка, в поле можно просто положить новый объект прямо в процессе работы. Даже во время компиляции наследование не очень гибко — в разных языках есть ограничения на количество классов-предков, всякое ромбовидное наследование и тонкости, с ним связанные, и т.д. Агрегация лишена этих недостатков. У агрегации тоже есть проблемы — во-первых, это чисто синтаксически необходимость явно описывать в нашем классе все методы, которые мы хотим поддерживать, и реализовывать их перенаправлением запроса агрегируемому объекту (опять-таки, больше строк кода). Во-вторых, с агрегацией не работает полиморфизм (хотя если сам агрегируемый объект полиморфный, можно добиться ряда занятных эффектов, см. паттерны “Состояние” и “Стратегия”).

Поэтому в целом можно сказать, что наследование имеет смысл использовать только для обеспечения полиморфизма. Наследование для переиспользования общей функциональности предка можно смело заменять на включение этого “предка” как поля во все классы, которым нужна его функциональность. А это, в свою очередь, значит, что наследоваться от конкретных (и даже абстрактных) классов нет никакой нужды — интерфейсов для обеспечения полиморфизма вполне достаточно, а переиспользовать код предка всё равно плохо. Из этого, в свою очередь, следует, что `private`-наследование, как оно реализовано в C++, суть ересь. И C++ настолько ужасен, что наследование в нём `private` по умолчанию. Впрочем, по моему опыту, `private`-наследование в реальном промышленном коде на C++ используется крайне редко.

На самом деле, заявления выше несколько категоричны. Есть способы использования наследования для обеспечения конфигурирования класса-предка — см., например, паттерн “Шаблонный метод” или большинство оконных библиотек. Там основная функциональность реализуется в классе-предке, потомки нужны, чтобы переопределить виртуальные методы предка так, чтобы всё вместе делало что-то полезное. Впрочем, большинство таких вещей писалось до массового распространения лямбда-функций в языках программирования, так что по-другому делать было просто очень неудобно. Возможно, мы скоро увидим фреймворки, которые наследованием не пользуются вовсе.

Наследование — по определению отношение “является” между типами. Есть простой способ проверки иерархии наследования на адекватность — принцип подстановки Барбары Лисков: любой код, который может работать с предком, должен мочь работать и с потомком. То есть наследование — это наследование интерфейса (в смысле обеспечения полиморфизма подтипов), причём интерфейса в широком смысле — обязательств и инвариантов.

Многие языки позволяют явно запретить наследование (`final`- или `sealed`-классы, `final`-методы). Есть две школы мысли по поводу того, как это использовать. Некоторые библио-

текописатели (например, Microsoft) считают, что в библиотечном коде запрещать наследование нельзя, потому что это существенно ограничивает гибкость. Где-то в документации от Microsoft приводился пример с подключением к базе данных, где наследование от библиотечного класса использовалось только для того, чтобы определить конструктор, представляющий нужный Connection String, и это типа удобно. Некоторые наоборот считают, что всё, для чего наследование не разрешено явно, должно быть закрыто для наследования, потому что если вы хотите позволить наследоваться от вашего класса, вам надо подумать о защите ваших инвариантов от потомков, о правильном использовании потомками методов вашего класса и т.д., это некоторая работа и делать её для каждого класса смысла нет.

Некоторые code smells, которые намекают на то, что с иерархией наследования что-то не так:

- базовый класс, у которого только один потомок; оставлять возможности для расширения “на будущее”, которое никогда не наступит, не стоит, благо в будущем можно и отнаследоваться, если будет надо, это не страшный рефакторинг;
- пустые переопределения — предок декларирует контракты, которые потомок не может содержательно выполнить;
- очень много уровней в иерархии наследования — обычно больше семи уровней означает, что всё совсем плохо, три-четыре уровня — подозрительно; бывают исключения, например, иерархия наследования в абстрактном синтаксическом дереве компилятора;
- новые методы с такими же именами, как у родителей — во-первых, это путает пользователя, во-вторых, это сильный индикатор того, что потомок вовсе потомком не является, а делает что-то своё.

Бывает, что наследование не используется там, где использоваться должно бы. Самый явный индикатор этого — switch или длинная цепочка из if-else, где в зависимости от значения какого-то поля выбирается какое-то действие. Это на самом деле попытка реализовать вручную таблицу виртуальных методов. Простой пример:

```
class Operation {  
    private char sign = '+';  
    private int left;  
    private int right;  
  
    public int eval() {  
        switch (sign) {  
            case '+': return left + right;  
            case '-': return left - right;  
        }  
        throw new RuntimeException();  
    }  
}
```

Это легко можно заменить на виртуальный вызов, где роль того поля, по которому делается выбор, играет тип времени выполнения объекта, у которого этот виртуальный вызов делается:

```
abstract class Operation {  
    private int left;  
    private int right;  
  
    protected int getLeft() { return left; }  
    protected int getRight() { return right; }  
    abstract public int eval();  
}  
  
class Plus extends Operation {  
    @Override public int eval() {  
        return getLeft() + getRight();  
    }  
}  
  
class Minus extends Operation {  
    @Override public int eval() {  
        return getLeft() - getRight();  
    }  
}
```

Опять-таки, кода становится больше, но архитектура в целом становится лучше — теперь нет switch-а, который вынужден знать про все возможные операции, набор операций несложно расширить, теперь они могут быть довольно сложными (вплоть до того, чтобы собираться из элементарных операций в рантайме, см. паттерн “Интерпретатор”). И операции в процессе развития программы могут получить ещё какую-нибудь ответственность, например, “печатать себя”, и это будет очень легко поддерживать, не городя ещё один гигантский switch.

## 3.4. Вопросы инициализации

Особого внимания заслуживают вопросы создания и инициализации объектов (и, что более интересно, сложных конструкций из нескольких объектов). Касательно конструкторов рекомендации такие:

- Инициализируйте все поля, которые надо инициализировать. Конструктор — это тот код, который более-менее обязательно выполняется при создании объекта, поэтому его задача — сделать так, чтобы все инварианты после него выполнялись. Надеяться на то, что пользователь честно-честно вызовет метод `init()` после конструктора — бессмысленно.
- Не вызывайте виртуальные методы из конструктора. Обычно вам даже компилятор скажет, что вы неправы, но если нет, последствия могут быть печальны. Дело в том,

что при создании объекта конструкторы вызываются по цепочке, соответствующей иерархии наследования, при этом сначала вызываются конструкторы предка. Так что если ваш любимый язык таки позволяет сделать виртуальный вызов, вызовется метод потомка ещё до того, как конструктор потомка будет иметь шанс установить свои инварианты.

- Используйте `private`-конструкторы для объектов, которые не должны быть созданы, или одиночек (некоторые языки позволяют явно сказать, что конструктора нет — = `delete`; в C++, `abstract` в C# и Java), используйте `protected`-конструкторы для абстрактных классов — создать их всё равно нельзя, так что нечего смущать пользователя, но из потомков вызывать конструктор может быть нужно.
- Одиночек надо использовать с большой осторожностью, потому как это замаскированные глобальные переменные со всеми вытекающими проблемами.
- Deep copy предпочтительнее Shallow copy, хотя второе может быть эффективнее. Пользователи в любом случае ожидают семантику глубокого копирования и могут быть удивлены, если изменения одного объекта приведут к изменению и другого. Глубокое копирование может быть сложным в реализации, если речь идёт про большой граф объектов, особенно если надо разбираться с идентичностью объектов (если объекты хранятся в БД или передаются по сети, их идентичность определяется не их местом в памяти, так что о ней надо думать при копировании). Если с идентичностью проблем нет, можно использовать такой хак — сериализовать копируемый объект в поток байт в памяти, потом десериализовать.

### 3.5. О мутабельности

Мутабельность — способность объекта изменять своё состояние — довольно важное свойство, оказывающее существенное влияние на архитектуру. Если кратко, то мутабельное состояние — это плохо и его надо стараться избегать. Более подробно, мутабельность запускает причинно-следственные связи в программе — кто-то вызвал метод, меняющий состояние, потом кто-то ещё, потом объект ведёт себя не так, как ожидалось. Кроме того, мутабельность создаёт дополнительные сложности для многопоточного программирования — если состояние можно изменять, надо следить, чтобы два потока не пытались его изменить одновременно.

Тем не менее, основные современные языки программирования предполагают, что состояние мутабельно по умолчанию, а иногда (например, в Java) надо ещё помучаться, чтобы сделать класс немутабельным. Вот про что надо помнить, делая немутабельный класс.

- Не предоставлять методы, модифицирующие состояние. Часто модификация состояния по смыслу всё-таки нужна, но есть полезный приём — заменить модифицирующие состояние методы на методы, возвращающие копию объекта. При этом, разумеется, поменяется семантика программы, так что это не чисто механическое изменение.
- Не разрешать наследоваться от класса.
- Сделать все поля константными.

- Не давать никому ссылок на поля мутабельных типов. Даже если поля константны, константное поле мутабельного ссылочного типа не позволяет только модифицировать саму ссылку, объект, на который она указывает — сколько угодно.

Правила хорошего тона говорят, что немутабельным должно быть всё, что может быть немутабельным без существенного вреда смыслу или скорости работы программы.

## 3.6. Об оптимизации

Кстати, по поводу скорости работы программы:

*Во имя эффективности (без обязательности ее достижения) делается больше вычислительных ошибок, чем по каким-либо иным причинам, включая непроходимую тупость.*

— William A. Wulf

*Мы обязаны забывать о мелких усовершенствованиях, скажем, на 97% рабочего времени: опрометчивая оптимизация — корень всех зол.*

— Donald E. Knuth

*Что касается оптимизации, то мы следуем двум правилам:*

*Правило 1. Не делайте этого.*

*Правило 2 (только для экспертов). Пока не делайте этого — т.е. пока у вас нет абсолютно четкого, но неоптимизированного решения.*

— M. A. Jackson

В общем, скорость работы программы не так важна, как её объясняющая способность. Часто она вообще не имеет значения — время реакции пользователя составляет сотни миллисекунд, так что 20 миллисекунд вы сортируете массив или 40, вообще не важно, если это надо сделать один раз по клику на кнопку. Для любителей спортивного программирования стремление по умолчанию писать “оптимальный” код может послужить причиной больших проблем с сопровождаемостью при довольно незначительном выигрыше в скорости работы. В общем, во-первых, надо себя сдерживать и стараться предпочитать простое решение быстро работающему (ну, до разумных пределов, всё-таки выбирайте алгоритм с подходящей асимптотикой, не надо числа Фибоначчи рекурсивно считать). Во-вторых, тонкой настройкой производительности (типа управления сборкой мусора и т.п.) надо заниматься *только* после исследования системы профилятором. Впрочем, есть противоположное оптимизации понятие — “пессимизация”, означающее откровенную глупость, приводящую к замедлению программы (типа случайного boxing-a внутри цикла в Java), этого тоже надо избегать.

## 3.7. Общие рекомендации

Наконец, общие рекомендации по поводу написания качественного кода, которые в другие разделы не попали:

- Fail Fast. Казалось бы, если программа падает, то это плохо. На самом деле, гораздо лучше, если программа упадёт, чем если продолжит работу в некорректном состоянии и испортит пользовательские данные. Поэтому принцип “Fail Fast” говорит, что программа должна завершить работу при малейшем подозрении на ошибку, а программист должен прикладывать усилия при написании кода, чтобы обнаружить

потенциальную ошибку как можно раньше. Например, сильно помогает активное использование assert-ов. Чем их больше, тем лучше — можно проверять инварианты, предусловия и постусловия методов, любые предположения и всё-всё-всё. Если есть языковые или библиотечные средства, позволяющие явно декларировать контракты или nullability, ими надо пользоваться (даже если это лишний код, замедляет рефакторинг и всё такое). Ещё полезный приём — “Защитное программирование”, когда ваш код не доверяет адекватности всего, что вовне — например, проверяет каждый параметр каждого public-метода на корректность. Не надо выключать проверки в релизной конфигурации — пусть у пользователя падает столь же часто, как у вас. Багрепорты помогут быстрее вычистить баги из системы.

- Документируйте все открытые элементы API. И заодно всё остальное, для тех, кто будет это сопровождать. Причём, желательно документировать подробно, не забывая те вещи, которые обычно забывают: предусловия и постусловия, бросаемые исключения, потокобезопасность, асимптотику.
- Статические проверки и статический анализ лучше, чем проверки в рантайме. Прежде всего, в этом помогает система типов, её надо использовать по полной.
- Юнит-тесты — чем больше, тем лучше (но не юнит-тестов вообще, а на интересные случаи).
- Continuous Integration, воспроизводимая сборка, запускающая юнит-тесты после каждого коммита в репозиторий.
- Не надо бояться всё переписать. Первые версии системы следует вообще проектировать на выброс, их цель — получить знания и опыт в предметной области. Если вы выкидываете код, знания всё равно остаются, так что работа выполнена не впустую. А вот иметь в “боевой” системе куски кода, которые были написаны когда-то давно и никто уже не знает зачем, может быть опасно — есть даже антипаттерн “Lava Flow”, описывающий как раз такую ситуацию, и единственное надёжное предлагаемое решение в этом антипаттерне — “переписать всё с нуля”.

# Лекция 3: Моделирование, UML

Юрий Литвинов

yurii.litvinov@gmail.com

## 1. Моделирование

Эта лекция начнёт рассказ о моделировании — важном инструменте для создания и, главное, описания архитектуры системы. Труд архитектора обычно находит внешнее выражение в архитектурной документации, ключевой составляющей которой являются визуальные модели программного обеспечения. Для создания визуальных моделей существует много визуальных языков, формальных и не очень, мы в этом курсе рассмотрим UML, с отвлечениями на некоторые другие визуальные языки.

Но начать следует с понятия “модель”. *Модель* — это упрощённое подобие некоторого объекта или явления, нужное для изучения некоторых его свойств, абстрагируясь от сложности того, что они моделируют. Модели используются повсюду — математические модели, реальные модели (например, модели самолётов для изучения их аэродинамических характеристик) и, конечно, модели в разработке программного обеспечения.

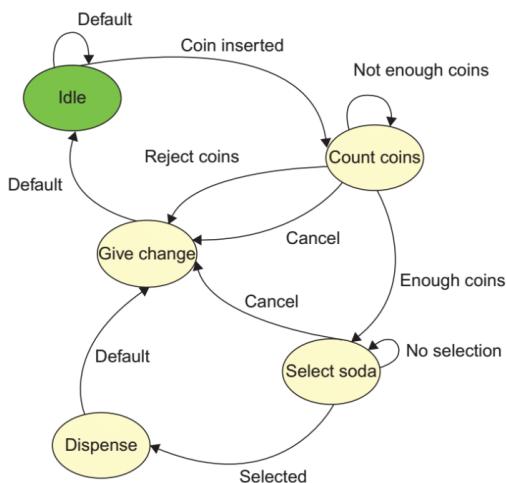
Модели принципиально содержат меньше информации, чем реальность. Нет смысла максимально точно моделировать какой-то объект, ведь если сложность не проблема, можно изучать и сам объект. А раз так, то каждая модель всегда создаётся для какой-то определённой цели, выделяя из моделируемого объекта или явления только те свойства, которые важны для исследования. В частности, поэтому при моделировании ПО рисовать диаграммы “вообще” в корне неправильно. Кроме того, модели субъективны. В случае с ПО архитектор сам решает, что следует показать на модели, а что нет. Это, в частности, нужно опять-таки для того, чтобы отделять существенные свойства от несущественных. И наконец, модели всегда ограничены — они в принципе не могут моделировать бесконечную сложность физических явлений или мельчайшие подробности программного кода. Так что стремиться к полноте при моделировании не только бессмысленно, но и просто нельзя, иначе модели станут бесполезны.

Кстати, есть хорошее высказывание про моделирование, более чем применимое к миру разработки ПО: “All models are wrong, some are useful”. Именно так, модели всегда неправильны (по вышеизложенным соображениям, они не могут содержать всю информацию о системе), но некоторые из них полезны. Поэтому моделирование часто не применяется при разработке ПО вовсе (потому что зачем, они всё равно врут, а создать и поддерживать модель — это большая работа). Поэтому, возможно, часть этого курса про моделирование конкретно вам никогда в жизни не пригодится. Но если всё-таки в вашей компании вдруг решат использовать визуальные модели, вы должны знать, что это, зачем и как это делать. И главное, чего ожидать — не точности и полноты, а именно полезности.

## 1.1. Модели в проектировании ПО

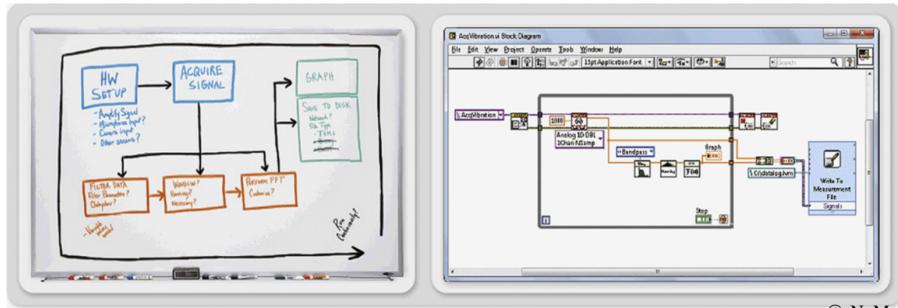
Полезность моделей при проектировании ПО заключается прежде всего в управлении сложностью. В предыдущей лекции говорилось, что сложность — это неотъемлемое свойство программного обеспечения, так что модели ПО, в отличие от математических или физических моделей, сами бывают довольно сложны. Но возможность взглянуть на систему с нескольких разных точек зрения, выделяя каждый раз разные существенные моменты, оказывается очень ценной при проектировании и документировании системы. На самом деле, модели ПО часто моделируют не только само ПО, но и окружение, в котором оно работает — будь то автоматизируемый бизнес-процесс или внешние системы или пользователи.

Вот небольшой пример модели из аналогичного курса университета Южной Калифорнии (на него мы далее тут будем неоднократно ссылаться с любезного согласия автора курса проф. N. Medvidovic):



Видно, что модель ПО может быть довольно неформальна (хотя на самом деле это конечный автомат, пожалуй, одна из самых формальных моделей), проста (хотя за каждым состоянием и переходом может скрываться очень много строк кода), и при этом полезна. Причём не только с иллюстративной, но и с pragматической точки зрения — можно отследить множество достижимых состояний, проверить, что автомат всегда может вернуться в исходное состояние. Это ценное свойство всех моделей ПО — они позволяют (в большей или меньшей степени, в зависимости от используемого формализма) понять, проанализировать и даже протестировать систему ещё до того, как будет написана первая строчка кода.

Используемые нотации и способы моделирования бывают разные и зависят от целей моделирования:



Самые ходовые модели, на самом деле, неформальные. Большая часть полезных моделей служит только для поддержки общения, такие модели рисуются на доске, их обсуждают и тут же стирают, в лучшем случае фотографируя. Более формальные модели используются в документации, там их некому вживую пояснить, поэтому требуется использовать более-менее стандартные языки моделирования (например, UML) — чтобы даже если систему отдали на аутсорсинг в Индию, архитектуру поняли бы однозначно. Самые формальные модели — исполнимые, это полноценные языки программирования, только в графическом синтаксисе (например, LabVIEW, модель на картинке, справа — на нём по-настоящему программируются микроконтроллеры и даже целые их сети).

## 1.2. Архитектурные модели

Как говорилось в первой лекции, архитектура — это набор основных решений, принятых для данной системы. *Архитектурная модель* — это некоторый артефакт, который отражает некоторые или все эти решения. Описание архитектуры может иллюстрироваться сразу многими архитектурными моделями, каждая из которых описывает свой набор важных решений. Какое из решений считать важным, решает архитектор.

Архитектурное моделирование — это процесс уточнения и документирования архитектурных решений. Модели делают решения явными и часто сами “задают вопросы” архитектору, позволяя лучше понять создаваемую архитектуру и заполнить пропущенные моменты. Лучше всего направляют архитектурную мысль языки, специально созданные для описания архитектуры (AADL, UML, IDEF, ...), так что правильный выбор нотации может быть критичен для успешности моделирования и разработки архитектуры вообще.

Однако существует опасность “архитектурного паралича” проекта, когда неопытная команда пытается настолько детально проработать архитектуру, что проект закрывается из-за нехватки денег ещё до того, как написана первая строчка кода. Чтобы этого избежать, надо определиться с тем,

- какие архитектурные решения нуждаются в моделировании;
- на каком уровне детализации;
- насколько формально.

Естественно, самые важные части архитектуры должны получать больше всего внимания при моделировании. Под самыми важными следует понимать не те решения, которыми вы особо гордитесь, а то, что затронет большое количество пользователей или

разработчиков системы. Не очень важным компонентам можно уделять меньше внимания — описывать их менее подробно и/или менее формально. Помните, что моделирование выполняется не ради высшего блага, а чтобы упростить разработку, поэтому при выборе, что моделировать, важно учитывать соотношение трудозатрат и выгоды. Следует помнить также про процесс поддержки моделей — при изменении архитектуры и кода диаграммы, возможно, тоже придётся перерисовывать, иначе они быстро потеряют всякий смысл. Стоимость создания и поддержания модели не должна быть больше преимуществ от её использования.

Немного подробнее о преимуществах, на которые можно надеяться при использовании моделирования. Модели — это:

- инструмент, направляющий и облегчающий проектирование — особенно если используется подходящий визуальный язык, модели сами подсказывают направление архитектурной мысли и делают очевидными слабые места архитектуры;
- средство коммуникации между разработчиками — “one picture worth a thousand words”, с помощью диаграмм проще донести идею решения, диаграммы проще обсуждать, чем код или идеи, выраженные только словами;
- наглядный инструмент для общения с заказчиком — бывают не очень формальные диаграммы, понятные неспециалистам. Проще (и выглядит более профессионально!) показать диаграммы, чем долго и утомительно рассказывать, что вы предлагаете;
- средство документирования и фиксации принятых решений — в этом качестве диаграммы используются в архитектурных описаниях, использование стандартных визуальных языков повышает вероятность, что идеи будут правильно поняты коллегами, даже если вы не можете лично их передать;
- исходник для генерации кода — бывают очень формальные диаграммы, по которым можно генерировать работающее приложение. Чаще, однако, так делать нет смысла, потому что между моделью и кодом есть так называемый *семантический разрыв*, связанный с тем, что модель содержит принципиально меньше информации, чем надо для того, чтобы её можно было исполнить. Например, любой нормальный инструмент для рисования UML-диаграмм умеет генерировать заглушки классов (объявления классов и методов, даже с полями, но без реализаций), но это не то чтобы суперполезно, потому что сгенерированный код потом всё равно надо дописывать руками и поддерживать (и приводить к конкретно вашему стайлгайду). Так что в общем случае на возможность сгенерировать код по моделям лучше особо не рассчитывать.

### 1.3. Виды моделей

Рассмотрим, какого рода модели вообще используются при проектировании программного обеспечения.

### 1.3.1. Естественный язык

Самый распространённый вид моделей, который обычно даже моделями-то не считается — это просто текст на естественном языке. Вот простой пример текстовой модели игры “Посадка на луну” из лекции проф. N. Medvidovic:

*“The Lunar Lander application consists of three components: a **data store** component, a **calculation** component, and a **user interface** component.*

*The job of the **data store** component is to store and allow other components access to the height, velocity, and fuel of the lander, as well as the current simulator time.*

*The job of the **calculation** component is to, upon receipt of a burn-rate quantity, retrieve current values of height, velocity, and fuel from the data store component, update them with respect to the input burn-rate, and store the new values back. It also retrieves, increments, and stores back the simulator time. It is also responsible for notifying the calling component of whether the simulator has terminated, and with what state (landed safely, crashed, and so on).*

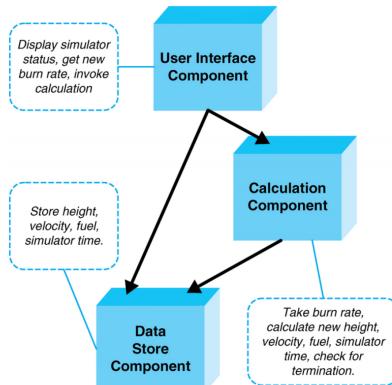
*The job of the **user interface** component is to display the current status of the lander using information from both the calculation and the data store components. While the simulator is running, it retrieves the new burn-rate value from the user, and invokes the calculation component.”*

© N. Medvidovic

Преимущества моделей на естественном языке очевидны — чтобы читать такие модели, не требуется учить синтаксис новых языков, язык очень выразителен (в буквальном смысле позволяет описать все мыслимые модели), максимально гибок. Недостатки, впрочем, тоже очевидны — естественный язык неформален, не строг, слишком многословен, бесполезен для автоматической обработки (в этом плане последние достижения в NLP дают некоторую надежду, но пока что алгоритмы разбора текстов на естественных языках очень далеки от применения в инструментах разработки архитектуры).

### 1.3.2. Неформальные графические модели

Активно используются и неформальные графические модели — диаграммы, рисуемые в PowerPoint, InkScape, Visio (без плагина UML) и подобных редакторах. На самом деле, такие диаграммы, скорее всего, встречаются в мире разработки ПО даже чаще, чем UML-диаграммы:



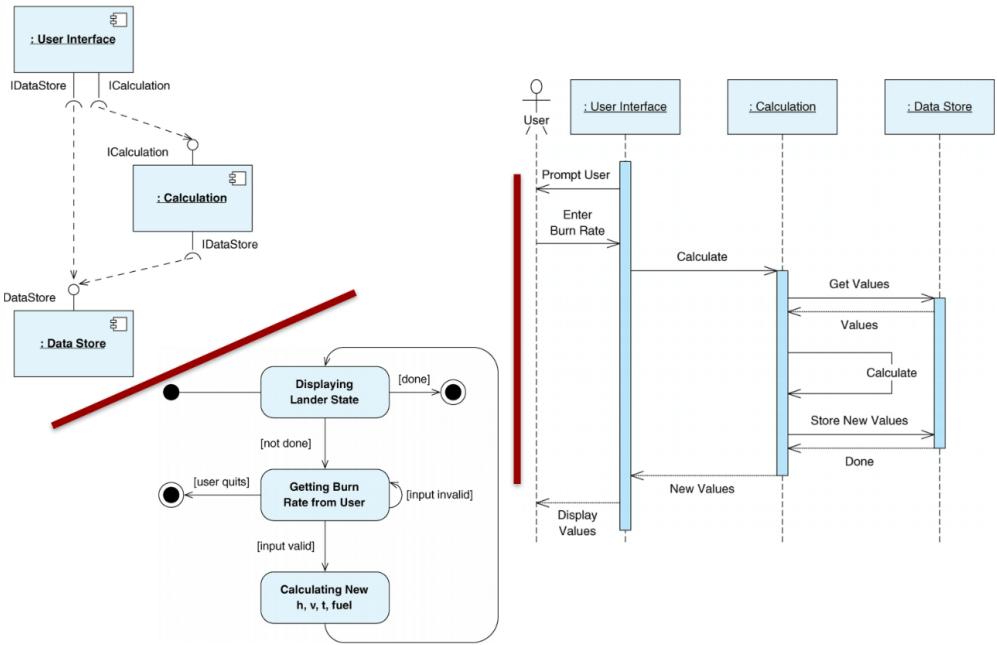
© N. Medvidovic

Преимущества такого подхода — простота понимания (опять-таки, не требуют специальных знаний), очень гибкая нотация, их можно сделать чисто внешне красивыми. Недостатки — опять-таки неформальность, нестрогость и неоднозначность, и тут, в отличие от текстовых моделей, есть подводный камень — неформальные диаграммы часто воспринимаются неспециалистами как формальные и строгие. Такие диаграммы всё ещё практически бесполезны для автоматической обработки. Фигуры и линии с их координатами обычно несложно достать из файлов сохранения, но потом их приходится мучительно парсить, чтобы программно понять смысл нарисованного. Например, в Visio фигура и текст, который внутри написан, могут быть формально никак не связаны и даже храниться в разных частях файла.

### 1.3.3. Формальные графические модели

Формальные графические модели — это прежде всего модели на формальных языках типа UML, SysML, Entity-Relationship, IDEF0 и т.д., таких языков довольно много. Такие языки часто состоят из нескольких разных нотаций (или “диаграмм”), позволяющих отдельно моделировать разные точки зрения на систему, стандартны — в чём их главное преимущество перед способами, рассмотренными выше — и, поэтому, имеют хорошую инструментальную поддержку (например, существует больше десятка только популярных UML-инструментов для создания архитектурных моделей, непопулярных тысячи). Важным достоинством является формальность языков, что может быть непривычно для людей, привыкших воспринимать UML как картинки — у UML и у любого другого такого языка есть строгий синтаксис, описывающий множество корректных диаграмм, как и у текстовых языков программирования. Важный недостаток формальных графических моделей — частое отсутствие строгой семантики (в UML из 14 видов диаграмм формальная семантика хоть как-то описана только для двух, хотя синтаксис подробно описан для каждой). Кроме того, поскольку такие языки поддерживают много точек зрения, между ними сложно обеспечить консистентность, сложно расширять языки и модифицировать их под свои нужды.

Вот пример трёх разных диаграмм (или “подъязыков”) UML:



© N. Medvidovic

### 1.3.4. Формальные текстовые языки

Даже формальные графические языки недостаточно формальны (обычно не имеют исполнимой семантики, чтоinneудивительно, потому что это языки моделирования всё-таки). Поэтому существуют и формальные текстовые языки описания архитектуры, например, AADL (может быть, VHDL можно отнести к таким языкам, хоть он слишком специфичен). Это языки, где компоненты системы и интерфейсы между компонентами описываются формально на языке программирования (но без реализации, конечно), выпи-сываются и доказываются различные утверждения про моделируемую систему (например, ограничения на время отклика, пропускную способность). Пример описания архитектуры на AADL от проф. N. Medvidovic:

```

data lander_state_data
end lander_state_data;
bus lan_bus_type
end lan_bus_type;

bus implementation lan_bus_type.ethernet
properties
    Transmission_Time => 1 ms .. 5 ms;
    Allowed_Message_Size => 1 b .. 1 kb;
end lan_bus_type.ethernet;
system calculation_type
features
    network : requires bus access
        lan_bus.calculation_to_datastore;
    request_get : out event port;
    response_get : in event data port lander_state_data;
    request_store : out event port lander_state_data;
    response_store : in event port;
end calculation_type;

system implementation calculation_type.calculation
subcomponents
    the_calculation_processor :
        processor calculation_processor_type;
    the_calculation_process : process
        calculation_process_type.one_thread;

```

© N. Medvidovic

Используются такие языки в основном там, где требуется высокая надёжность и некоторые гарантии на параметры системы — во встроенных системах, системах реального времени и т.д., они же используются часто при проектировании программно-аппаратных и аппаратных систем.

Основное преимущество подобного подхода — возможность формального анализа свойств системы ещё до начала реализации, и для AADL существуют весьма продвинутые инструменты анализа. Основные недостатки — слишком большая многословность и детальность (уже не нарисовать на доске), сложность в изучении и использовании (поэтому в результате одного опроса IT-компаний Турции выяснилось, что половина программистов слыхом не слыхивала про AADL).

## 1.4. Ещё о визуальных моделях

С архитектурными моделями связано ещё несколько понятий, которые мы пока подробно не рассматривали.

- Метафора визуализации — это договорённость о том, как будут представляться сущности языка. Например, в UML классы представляются в виде прямоугольников, разделённых на три области (имя класса, поля и методы), а в нотации Буча классы представлялись в виде облаков, нарисованных пунктирными линиями (а объекты — в виде облаков, нарисованных сплошными линиями; ну а что, классы — это что-то абстрактное). Поскольку программное обеспечение незримо, метаформа визуализации — не более чем договорённость между программистами.
- Точка зрения моделирования — какой аспект системы и для кого моделируется. Наличие такого понятия связано с тем, что модель принципиально проще моделируемой системы, так что приходится выбирать, какие детали оставить за её рамками.

Бывают модели для программистов, по которым они должны понять, какие классы и с какими методами надо реализовать, бывают — для менеджеров, чтобы те могли видеть, сколько уже реализовано и сколько ещё осталось, бывают — для заказчиков, чтобы объяснить им, что в итоге получится и почему столько стоит. Перед тем, как рисовать диаграмму, важно понять, для кого и зачем мы её рисуем. Не следует рисовать диаграммы просто потому что мы можем.

- Семантический разрыв — принципиальная неспособность модели полностью специфицировать систему; разрыв между информацией, содержащейся в модели и информацией, необходимой, чтобы компьютер мог выполнить программу, которую эта модель описывает. Впрочем, семантический разрыв — это не приговор, исполнимые модели бывают и активно используются (впрочем, если быть точными, исполнимая модель — это не модель, а программа на графическом языке). Чаще всего, однако, в реальной жизни встречаются одноразовые модели — нужные только для того, чтобы передать идею. Их не то что нельзя исполнить, их часто нельзя даже понять без помощи автора.

## 2. UML

Unified Modeling Language (UML) — пожалуй, самый популярный визуальный язык, используемый для разработки программного обеспечения, стандарт де-факто архитектурного моделирования. Пользуются им далеко не все проекты и компании, но знать его должен каждый уважающий себя программист и, тем более, архитектор, поэтому придётся потерпеть довольно нудный рассказ про UML далее.

UML не любят, в частности, из-за его сложности — на самом деле это не один язык, а порядка 14 разных языков, объединённых единым стандартом и единым описанием синтаксиса (то есть метамоделью). Обычно когда говорят UML, имеют в виду диаграммы классов, но это только один из 14 языков, бывают совсем другие нотации, описывающие систему с разных точек зрения.

UML разрабатывается консорциумом Object Management Group (OMG), в который входят крупные ИТ-компании и университеты (более сотни). OMG же стоит за такими стандартами, как CORBA и BPMN. Некоторые версии UML (1.4.2 и 2.4.1, не очень свежие, но good enough) приняты как международные стандарты ISO. Первая из получивших распространение стандартизованная версия UML имела номер сразу 1.1 и появилась в 1997 году, до неё была спецификация 0.9 (1996 год), обратите внимание, примерно в то же время, когда появились первые версии Java.

Появился UML не из воздуха: компания Rational наняла трёх ведущих методологов, занимавшихся визуальными языками и, собственно, методологией разработки ПО — Айвара Якобсона, Джеймса Рамбо и Гради Буча (того самого Гради Бucha, картинки из знаменитой книжки которого “Object-oriented analysis and design” были в предыдущей лекции). Каждый из этих уважаемых людей уже имел свою, и популярную, методологию проектирования. Буч был автором “метода Буча” и, кстати, сооснователем компании Rational, Рамбо был автором Object Modeling Technique (откуда в UML пришли диаграммы классов по сути), Якобсон — автор Object-oriented software engineering, откуда в UML пришли диаграммы случаев использования. Была создана не только единая нотация, призванная покончить

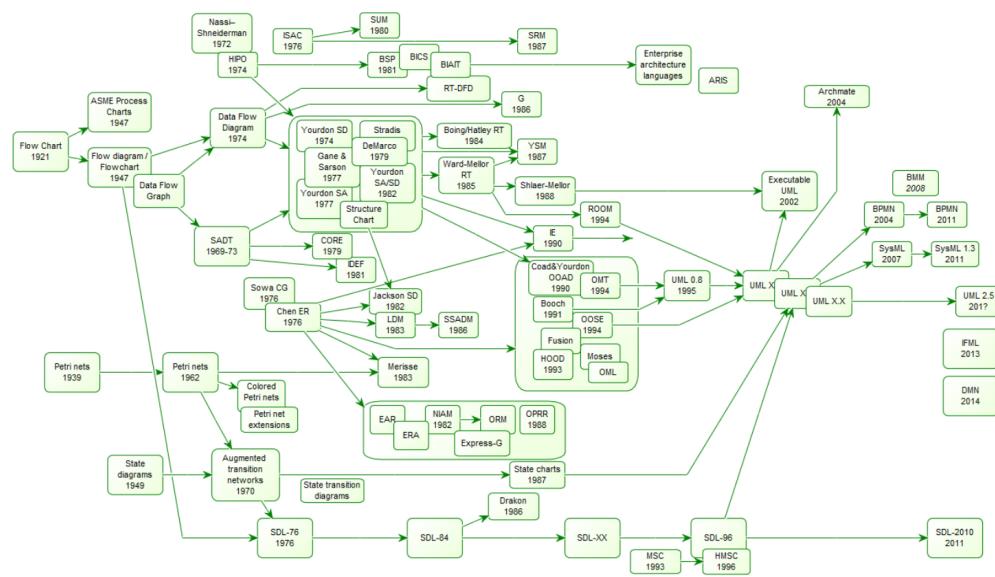
с языковой разрозненностью в области архитектуры в начале 90-х, но и единая методология разработки ПО: Rational Unified Process (RUP), бывшая популярной в конце девяностых, но побеждённая Agile-методологиями в начале двухтысячных. RUP предполагал активное использование UML, естественно. И так уж получилось, что компания Rational была первой на рынке с инструментами поддержки UML и RUP (кто бы мог подумать), стала фактическим монополистом в конце 90-х и, наверное, неплохо на этом заработала.

Дальнейшее развитие UML имеет одну серьёзную веху — UML 2.0 (2005 год), когда язык существенно переработали и приспособили не только для разработки программного обеспечения, но и для аппаратных систем тоже (используется он до сих пор всё-таки в основном программистами, у инженеров и так всё хорошо было с визуальными нотациями и их инструментальной поддержкой). Нотация языка тоже существенно поменялась (например, диаграммы активностей и диаграммы конечных автоматов сделали двумя отдельными языками), так что это иногда приводит к путанице — в интернетах полно диаграмм в старой нотации. Актуальная на данный момент версия UML — 2.5.1, принятая в декабре 2017 года.

В UML определён механизм стандартных расширений, даже целых два — *профили* и *метамоделирование*. Профили — механизм легковесного расширения, позволяют уточнить уже существующую нотацию UML для использования в какой-то конкретной предметной области. Метамоделирование позволяет взять стандарт UML и его подредактировать, добавив новые элементы или убрав существующие. Это гораздо более трудоёмко, чем профили, и не все инструменты поддерживают такой подход, тогда как профили более-менее широко поддержаны. Тем не менее, и профили, и метамоделирование активно используются самим консорциумом OMG, так что у UML появились языки-“родственники” (например, SysML), которые по сути расширения UML через метамоделирование. Есть и куча стандартных профилей UML, но мы не будем в этом курсе на них останавливаться.

## 2.1. История UML

Вот замечательный рисунок из <https://modeling-languages.com/history-modeling-languages-one-picture-j-p-tolvanen/>, показывающий взаимосвязь между разными визуальными языками:



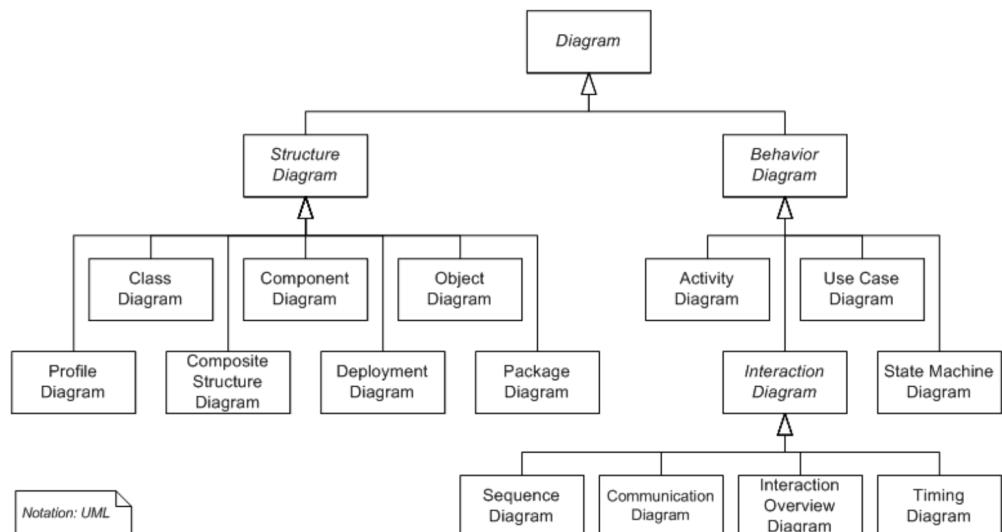
Как видно из картинки, всё началось ещё до появления электронных компьютеров, различного рода диаграммы использовались давным-давно. Однако серьёзная потребность в моделировании при разработке ПО возникла только в конце 60-х годов, когда уже появились высокоуровневые языки программирования и программы начали становиться сложнее рассчётов по формуле. В это время появляются нотации, копорые живы до сих пор: Data Flow Diagram, SADT (Structured Analysis and Design Technique), диаграммы Насси-Шнейдермана (из которых потом появился довольно известный нынче язык Scratch). На конец 1970-х приходится первый активный всплеск интереса к визуальным языкам и методологиям, появляются нотации SDL (Specification and Description Language), Entity-Relationship, IDEF, все они более чем живы до сих пор (SDL стандарт де-факто при описании телекоммуникационных протоколов, ER — при проектировании баз данных, емейство языков IDEF применяется в анализе бизнес-процессов).

Второй пик развития приходится на конец 80-х — начало 90-х (поиски серебряной пули и языков 4-го поколения, которые повысили бы продуктивность труда программиста так же, как в своё время языки высокого уровня повысили продуктивность по сравнению с ассемблером). Каждая уважающая себя ИТ-компания считала своим долгом создать свою уникальную технологию на основе визуальных языков, получить огромный прирост производительности (в этом никто особо не преуспел) и ни с кем не поделиться секретом. Тут появляются языки, которые потом лягут в основу UML (нотация Буча, OOSE, OMT), языки, которые войдут в UML как часть, но продолжат развиваться независимо — SDL, State charts (диаграммы конечных автоматов — казалось бы, общезвестная вещь, но formalизованы они были Харелом только в 1987 году). Почётного упоминания заслуживает ДРАКОН (Дружественный Русский Алгоритмический язык, Который Обеспечивает Наглядность) Паронджанова, язык G, который до сих пор используется в LabVIEW и средах на её основе — Robolab и, внезапно, стандартной среде для программирования роботов Lego. Андрей Николаевич Терехов создавал редакторы SDL и технологию RTST примерно в это же время.

В 1995 году появляются первые нестандартизованные спецификации UML, которые оканчивают эпоху языкового зоопарка, дальше UML плавно развивается до версии 2.0, которая вышла в 2005 году, это дало толчок к развитию кучи профилей UML или языков, родственных UML — Executable UML, BPMN, SysML и т.д. Параллельно развивались языки описания архитектур предприятий, и также в начале-середине двухтысячных стали модны предметно-ориентированные визуальные языки. Однако постепенно интерес к визуальным языкам пошёл на спад. Отчасти это произошло из-за Agile Manifesto, где было написано, что документация — это хорошо, но работающий код лучше; диаграммы не являются работающим кодом, семантический разрыв и всё такое. Отчасти — из-за того, что визуальные языки так и не показали обещанного прироста продуктивности, хайп угас и стало понятно, что визуальные языки — это сложно, очень тяжело в инструментальной поддержке и, в общем-то, не очень нужно. Тем не менее, UML всё ещё активно применяется в индустрии (порядка половины компаний-разработчиков его так или иначе используют), другие визуальные языки тоже сохранили свои экологические ниши и нынче хайп имеет шансы вернуться с новой силой в контексте end-user programming, интернета вещей и т.п.

## 2.2. Диаграммы UML

Но вернёмся, собственно, к UML. Как уже говорилось, UML 2.5.1 состоит из 14 видов диаграмм, довольно слабо связанных друг с другом, вот картинка (из UML Specification), где все эти диаграммы перечислены:



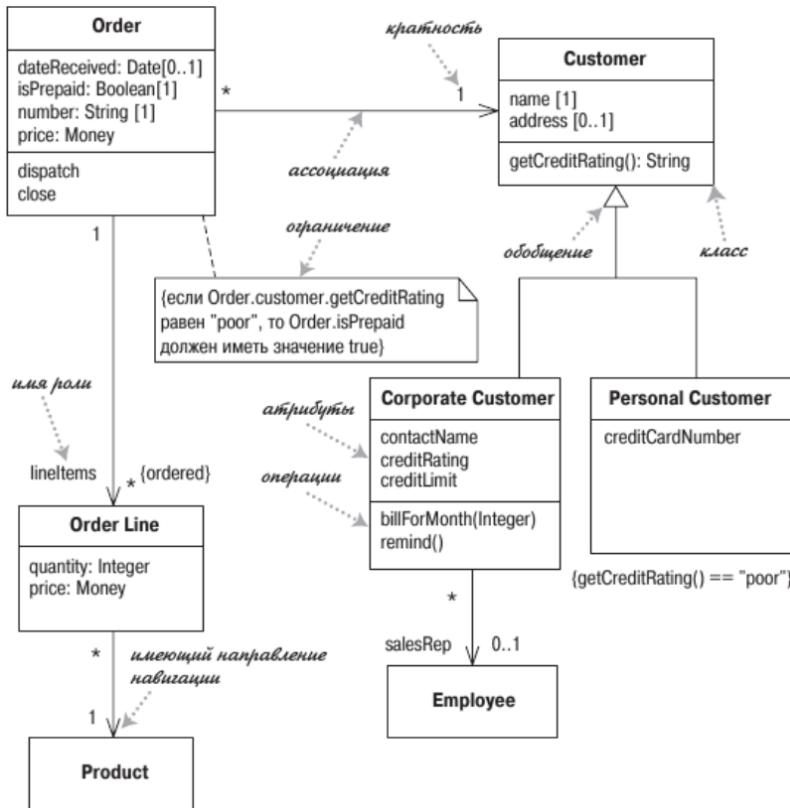
Сама картинка — это диаграмма классов UML, кстати.

Диаграммы делятся на две крупные категории — структурные и поведенческие. Структурные диаграммы показывают структуру системы времени компиляции — это прежде всего диаграмма классов, диаграмма компонентов, диаграмма пакетов и другие. Поведенческие диаграммы показывают, как система себя ведёт во время работы — это диаграммы состояний, диаграммы последовательностей, диаграммы активностей, сюда же относят

диаграммы случаев использования и другие, более специализированные диаграммы. Про почти все эти диаграммы будет подробно далее.

### 2.2.1. Диаграмма классов, общий синтаксис

Самая известная диаграмма UML — это, пожалуй, диаграмма классов. Её внешний вид и описание элементов из книжки М. Фаулера “UML. Основы” (кстати, книга очень рекомендуется как быстрая справка по UML, информации в которой вполне достаточно, чтобы успешно пользоваться языком):



© М. Фаулер. “UML. Основы”

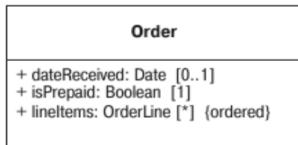
В общем-то, пояснений картинка не требует, надо только обратить внимание, что наследование в UML называется “обобщение” (generalization) и стрелка указывает на предка, а не на потомка (с этим часто путаются). Кратность ассоциации тоже может вызвать путаницу, она пишется у того конца ассоциации, к которому относится кратность. Например, Customer может иметь много Order (звёздочка означает “0 или больше”), но у каждого Order ровно один Customer.

Ещё одна важная особенность всех UML-диаграмм — это то, что они допускают не отображать некоторую информацию, если автор считает её не важной в данном контексте (вспомните про точку зрения моделирования). Например, на рисунке выше неверно, что

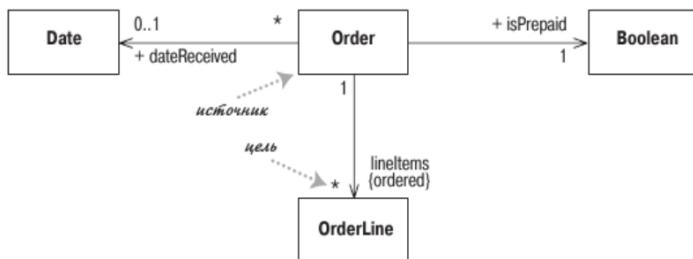
класс Employee не имеет ни методов, ни полей, просто автор посчитал, что они не важны и решил не загромождать диаграмму. То же касается и ассоциаций — отсутствие множественности или стрелок означает лишь, что они не специфицированы, они вполне могут быть выбраны самостоятельно программистом, который реализует систему.

## 2.2.2. Атрибуты

У диаграмм классов есть важная синтаксическая особенность — атрибуты и ассоциации представляют собой с точки зрения синтаксиса языка одно и то же, просто отображаются по-разному. Например, диаграммы

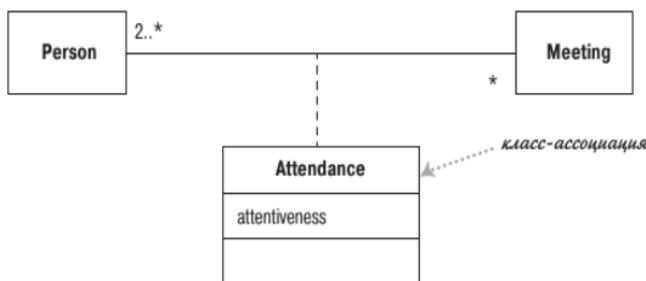


и



означают в точности одно и то же. Атрибуты обычно используются, когда связи между классами не важны: когда типы атрибутов — элементарные типы или перечисления (или даже структуры, короче, являются типами-значениями по смыслу; либо типы атрибутов — это полноценные классы, но из третьесторонних библиотек). Ассоциации — когда связи между классами важны для понимания архитектуры (чаще всего, когда типы атрибутов — классы из реализуемой системы).

Ассоциациям самим разрешается иметь атрибуты (хотя в реальных диаграммах это встречается довольно редко), рисуется это вот так:

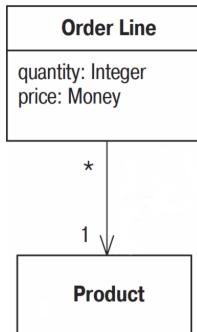


(все рисунки, как обычно, из книги М. Фаулера, “UML. Основы.”)

Описание свойства внутри класса имеет в самом общем виде форму “видимость имя: тип кратность = значение по умолчанию \{строка свойств\}”. При этом видимость бывает + (public), – (private), # (protected), ~(package). А кратность — 1 (ровно 1 объект), 0..1 (ни одного или один), \* (сколько угодно), 1..\*, 2..\*.

### 2.2.3. Атрибуты и код

Как диаграммы классов связаны с кодом лучше всего пояснить на примере. Если у нас есть такая диаграмма классов:



то по ней можно написать (или сгенерировать, по крайней мере большую часть) вот такой код (для примера, на Java):

```
public class OrderLine {
    private int quantity;
    private Product product;

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public Money getPrice() {
        return product.getPrice().multiply(quantity);
    }
}
```

Класс `Product` тут для краткости не показан, благо на диаграмме про него всё равно никаких подробностей не приводится. Обратите внимание, что навигабельная ассоциация стала полем, а поле `price` в ходе реализации превратилось не в поле, а в вычислимое свойство. Такие преобразования тоже вполне допустимы, если программист-реализатор считает, что так удобнее.

С двунаправленными ассоциациями несколько хитрее, потому что при реализации требуется ещё некий механизм поддержания целостности обоих концов ассоциации. Например, диаграмма



могла бы быть реализована следующим образом (теперь на C#, для разнообразия):

```
class Car {
    public Person Owner {
        get { return _owner; }
        set {
            if (_owner != null)
            {
                _owner.friendCars().Remove(this);
            }

            _owner = value;
            if (_owner != null)
            {
                _owner.friendCars().Add(this);
            }
        }
    }
    private Person _owner;
}

class Person {
    public IList Cars {
        get { return ArrayList.ReadOnly(_cars); }
    }

    public void AddCar(Car arg) {
        arg.Owner = this;
    }

    private IList _cars = new ArrayList();

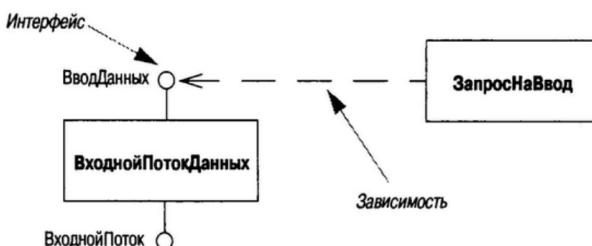
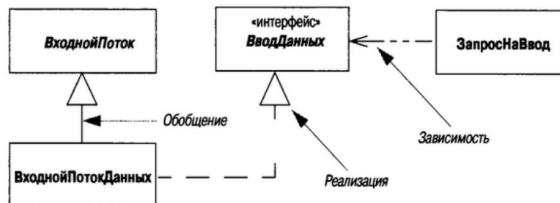
    internal IList friendCars() {
        // должен быть использован только Car.Owner
        return _cars;
    }
}
```

В C# нет аналога ключевому слову `friend` из C++, так что приходится использовать метод с пакетной видимостью, чтобы ограничить его использование. Но общий смысл тут

такой, что при изменении одного конца ассоциации надо не забыть поменять и объект на другом конце — сказать ему, что мы что-то добавили или что-то удалили.

## 2.2.4. Интерфейсы

На диаграмме классов есть целых два способа изображать интерфейсы:



© М. Фаулер. “UML. Основы”

Так же, как в Java, реализация интерфейса и наследование от класса в UML суть разные вещи, реализация интерфейса рисуется пунктирной линией. Либо используется “леденцовая” нотация, когда реализуемый классом интерфейс рисуется как кружок с названием интерфейса, соединённый с классом линией. Такая нотация предпочтительней, если нам не очень важны методы интерфейса либо если интерфейсов много. Нотация со словом `<<интерфейс>>` (или `<<interface>>` по-английски) используется, когда надо показать и методы интерфейса тоже (и тогда они пишутся как методы класса). Кстати, слова в ‘ёлочках’ называются в UML *стереотипами* и используются очень часто в самом языке и в механизме расширений через профили, когда придумывать новую фигуру для новой сущности не хочется (например, интерфейс — это почти класс, так что он рисуется как класс, просто над ним пишется, что он интерфейс).

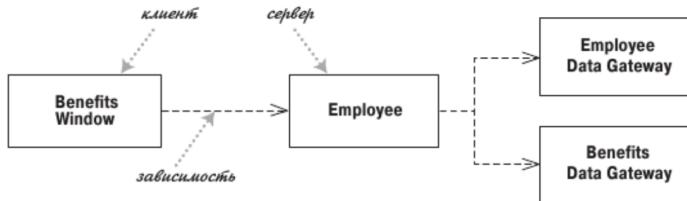
## 2.2.5. Зависимости

Потребление интерфейса рисуется как зависимость, пунктирной стрелкой. Зависимость — наиболее общий вид связи между классами, она означает просто, что один класс (или интерфейс) что-то знает про другой (например, он ему нужен, чтобы скомпилироваться). Зависимости могут быть уточнены стереотипами, некоторые из которых прописаны в стандарте:

- call — один класс вызывает метод другого;

- create или instantiate — один класс создаёт в каком-то из своих методов объект другого;
- derive — один класс представляет значение, которое может быть вычислено по другому;
- realize — один класс реализует другой (например, абстрактный класс);
- responsibility — контракт, который класс обязуется выполнять;
- refine — зависимость, которая может быть установлена даже между элементами с разных диаграмм, один элемент является уточнением другого — например, класс из модели предметной области уточняется набором классов из реализации;
- trace — зависимость, которая может быть установлена даже между элементами с разных диаграмм, означает, что один элемент как-то влияет на другой — например, случай использования может быть связан отношением trace с классами, которые его реализуют. Нужны такие зависимости прежде всего для автоматического отслеживания изменений.

Рисуются зависимости вот так:



© М. Фаулер. “UML. Основы”

Если надо уточнить зависимость, то уточняющее слово (call, create и т.д.) пишется в «ёлочках» над линией по центру.

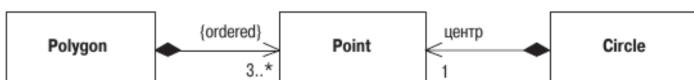
## 2.2.6. Агрегация и композиция

Агрегация и композиция — дальнейшие уточнения ассоциации. На самом деле, если на диаграмме нарисована просто ассоциация, то при реализации мы вправе выбрать, агрегацию или композицию использовать. Разница между агрегацией и композицией — во владении объектами, которые участвуют в ассоциации. Агрегация не предполагает владения:



Человек и клуб могут существовать независимо друг от друга, и когда закрывается клуб, его члены продолжают существовать (если это не тоталитарная секта). На диаграмме выше нарисовано, что в клубе может состоять несколько человек, и человек может состоять в нескольких клубах, при этом клуб знает о человеке, но не владеет им.

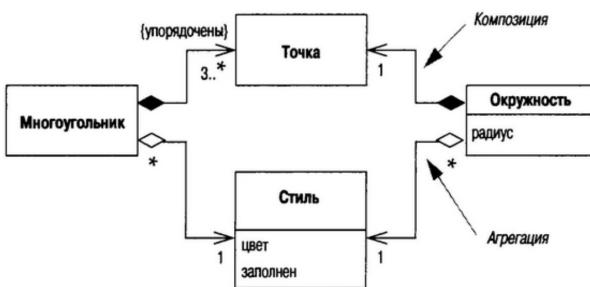
Композиция говорит, что один объект владеет другим объектом, то есть время их жизни связано и “хозяин” отвечает за удаление подчинённого ему объекта. Например, если мы пишем графический редактор, вполне может быть, что мы сделаем так:



Здесь многоугольник состоит из точек, и когда мы удаляем многоугольник, удаляются и все точки, которые его определяют. Круг тоже имеет одну точку — центр круга, и тоже, удаляем круг — удаляется и его центр. Так что и круг и многоугольник владеют своими точками, но у каждой точки вовремя исполнения может быть только один хозяин.

Агрегация и композиция важны, если в итоге мы будем писать на C++, там они помогают следить за тем, кто в итоге должен освободить память из-под объекта. Для языков со сборкой мусора это не так важно (хотя и полезно знать, кто кем владеет), поэтому агрегация с композицией используются в диаграммах классов довольно редко — обычно архитекторы ограничиваются ассоциациями, не специфицируя, агрегация эта конкретная ассоциация или композиция.

С точки зрения нотации надо запомнить, что агрегация — более слабое отношение, чем композиция, поэтому и ромбик в случае агрегации не закрашен. И ромбики рисуются около контейнера (то есть клуб содержит людей, а не наоборот) — можно понимать ромбик как оперение стрелы, указывающей от контейнера к содержащемуся в нём объекту. Ещё один небольшой пример:

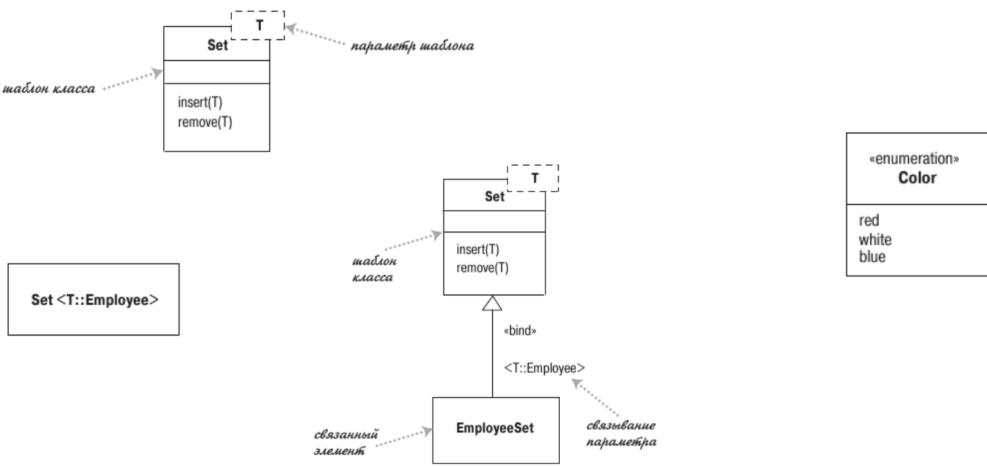


© М. Фаулер. “UML. Основы”

Многоугольник и окружность владеют своими точками, но не владеют стилями — у каждой геометрической фигуры должен быть стиль, но стили существуют независимо и могут переиспользоваться между фигурами. Точки между фигурами переиспользованы быть не могут.

## 2.2.7. Шаблоны и перечисления

В отличие от первых версий Java, в UML синтаксис для шаблонных классов (или генериков) был всегда: параметры-типы перечисляются через запятую в пунктирном прямоугольнике в правом верхнем углу класса. Инстанцирование шаблона (то есть подстановка фактических параметров-типов вместо формальных) рисуется либо как просто класс, у которого указано, что куда подставляется, либо как наследование со стереотипом <>bind>, как на рисунке ниже:



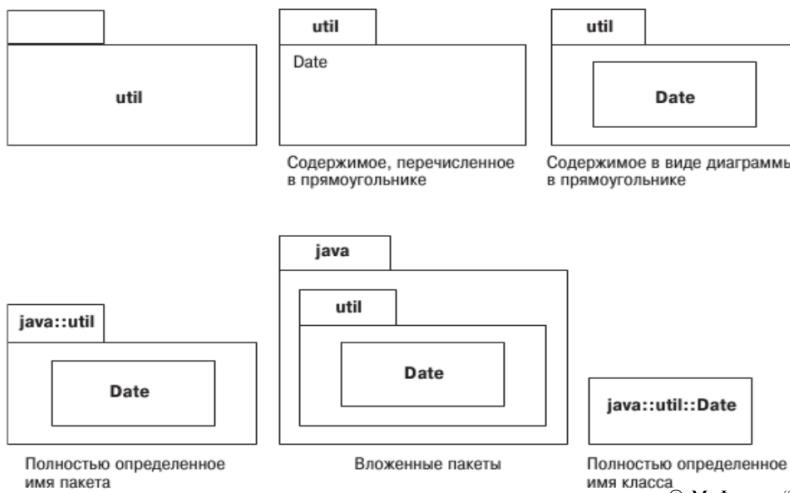
© М. Фаулер. “UML. Основы”

Второй способ используется, когда экземпляр шаблона доопределяет ещё какие-нибудь методы и поля (например, классы `QList<T>` и `QStringList` в библиотеке Qt, второй является специализацией первого для типа `QString`, но добавляет ещё специфичные методы, например, `join()`).

На рисунке также видно, как задаются перечисления. Используется стереотип `<>enumeration` и просто перечисляются элементы.

### 2.3. Диаграммы пакетов

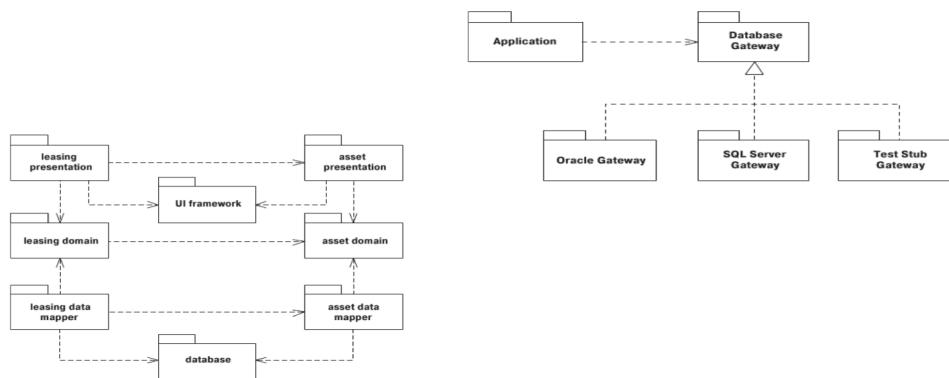
Следующий вид диаграмм UML — диаграммы пакетов. Это тоже структурные диаграммы, нужны они для того, чтобы показать разбиение кода по пакетам, либо сгруппировать по пакетам саму модель. “Пакет” в UML означает более-менее то же, что пакет в Java или пространство имён в C++. Синтаксис диаграммы пакетов очень простой:



© М. Фаулер. “UML. Основы”

На диаграммах пакетов также могут рисоваться компоненты и классы, которые в этих пакетах находятся. Это ещё одна не вполне очевидная особенность UML — диаграммы не являются отдельными языками, более того, в спецификации UML разбиения на диаграммы нет. Язык описывается единой метамоделью (сгруппированной по пакетам, кстати), а какие элементы на каких диаграммах рекомендуется рисовать, описано только в приложении к стандарту. Так что теоретически всё что угодно можно рисовать где угодно, хотя правила здравого смысла иногда этому мешают. Например, рисовать сущности структурных и поведенческих диаграмм на одной диаграмме нельзя, потому что это может сильно запутать читателя.

Диаграммы пакетов полезны ещё тем, что на них можно визуализировать зависимости между пакетами:



© М. Фаулер. “UML. Основы”

Просто зависимость означает, что пакет как-то связан с другим пакетом (так же, как зависимость между классами), стрелка “реализация” означает, что в пакете-предке есть хотя бы один класс, являющийся предком хотя бы одного класса из пакета-потомка, либо интерфейс, который пакет-потомок реализует. На самом деле, часто отношение реализации рисуют в том случае, если смысл пакета-предка в архитектуре — содержать интерфейсы или абстрактные классы, которые должны реализовывать другие пакеты.

## 2.4. Диаграммы объектов

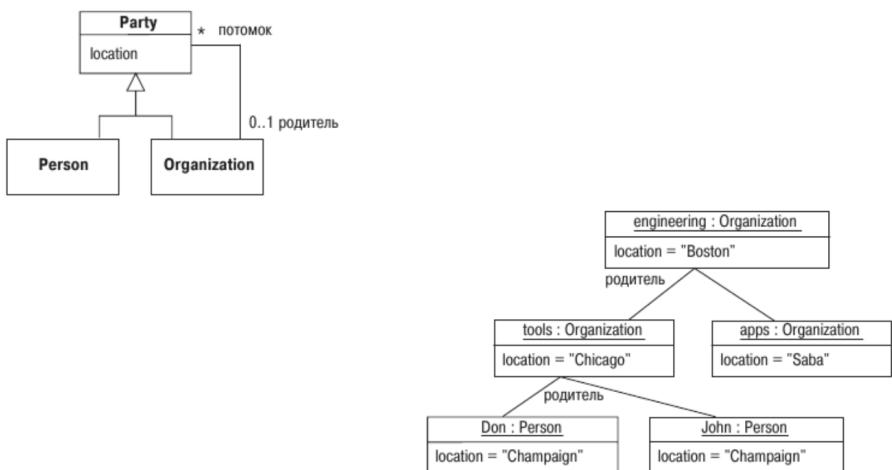
Диаграммы объектов визуализируют объекты в памяти во время работы программы и отношения между ними (так что по смыслу относятся к поведенческим диаграммам, но все их всё равно считают структурными, поскольку они описывают структуру времени выполнения, а не характер взаимодействия объектов). Диаграммы объектов можно понимать как снимок в какой-то определённый момент времени состояния системы, при этом множество всех возможных состояний описывается диаграммой классов.

Диаграммы объектов используют в двух случаях:

- чтобы визуализировать и проиллюстрировать диаграммы классов (как на рисунке ниже, по диаграмме классов не сразу очевидно, что она описывает дерево, а по диаграмме объектов — сразу);

- чтобы разобраться во взаимосвязях реально существующих объектов предметной области, ещё до того, как создана первая диаграмма классов (и тогда диаграмма классов будет на самом деле наведением классификации на объекты, выделенные на диаграмме объектов). Обращаю внимание на этот случай использования, мы увидим примеры применения диаграмм объектов в качестве инструмента анализа предметной области, когда дойдём до методологии предметно-ориентированного проектирования.

Синтаксис диаграмм объектов показан на этом рисунке:



© М. Фаулер. “UML. Основы”

Объект рисуется как класс, только его имя подчёркивается, указывается его тип, и каждому атрибуту ставится в соответствие значение.

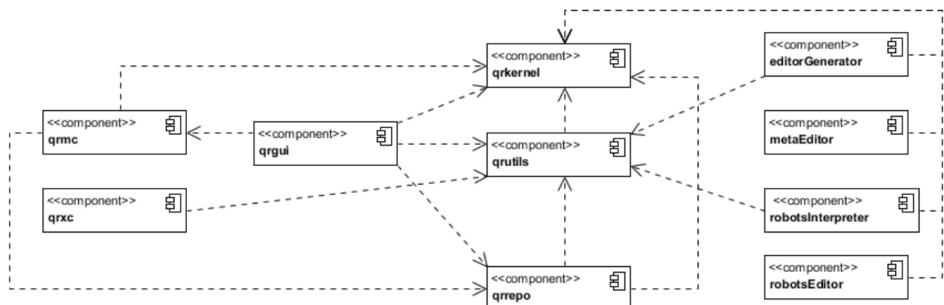
## 2.5. Диаграммы компонентов

Диаграммы компонентов — пожалуй, самые полезные для архитектора диаграммы UML. На них изображаются компоненты, из которых состоит система или подсистема, и взаимосвязи между ними. Точного определения термина “компонент” не существует, но все интуитивно представляют, что это такое — нечто структурно связанное и больше, чем класс. Компонентами могут быть пакеты, пространства имён, сборки, .dll/.so-файлы, отдельные веб-сервисы в распределённом приложении и т.д. В общем, это именно то, из чего состоит высокоуровневая архитектура приложения.

Почему диаграммы компонентов важнее и полезнее диаграмм классов — они переживают большую часть рефакторингов. Диаграммы классов придётся перерисовывать каждый раз, когда кто-то поменял название какого-нибудь класса или даже метода, добавил поле и т.д. Диаграммы компонентов неизменны до глобальной переработки архитектуры, которая в типичных проектах происходит только раз в несколько лет, так что у диаграмм компонентов много шансов сохранить свою актуальность в долгосрочной перспективе.

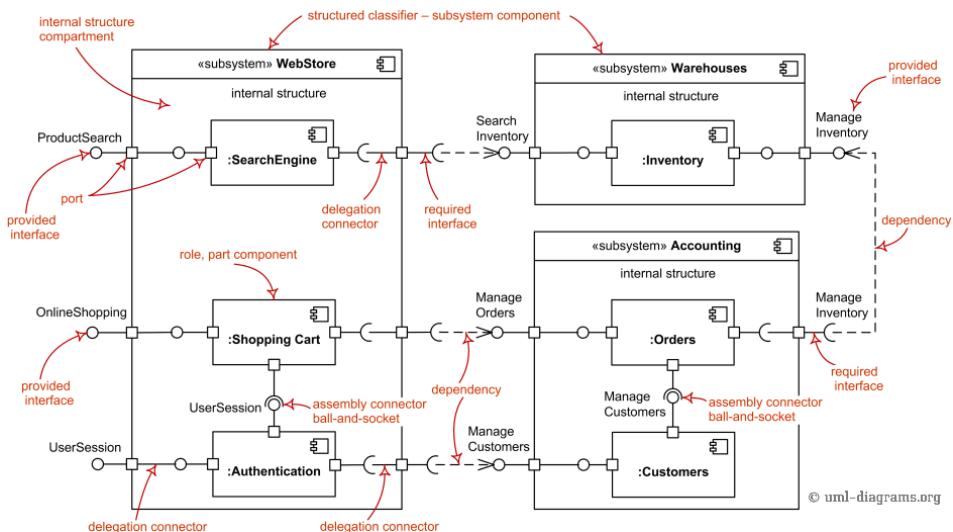
Полезны диаграммы компонентов как вид “с высоты птичьего полёта” на создаваемую систему. Например, вот диаграмма, описывающая высокоуровневую архитектуру проекта

QReal (<https://github.com/qreal/qreal>, инструмент для создания визуальных предметно-ориентированных языков и работы с ними):



Связи между компонентами тут означают зависимости по сборке, каждый компонент — это отдельный проект, который собирается в отдельный .dll/.so-файл. По такой диаграмме можно очень быстро рассказать общую архитектуру проекта: qrkernel ни от кого не зависит, там находятся классы, общие для всей системы; qrutils содержит полезную функциональность, используемую в других модулях; qrrepo — репозиторий, где хранятся данные, с которыми QReal работает; qrui — пользовательский интерфейс; qrmc и qrxc — это компиляторы визуальных языков, с которыми работает Real; editorGenerator, metaEditor, robotsInterpreter, robotsEditor — это плагины инструментальной поддержки работы с визуальными языками. Конечно, это не полноценное архитектурное описание, но даже этого достаточно, чтобы было понятно, куда смотреть в исходниках.

Вот описание синтаксиса диаграмм компонент, на сей раз с сайта <http://www.uml-diagrams.org> (очень рекомендую, как быструю справку по синтаксису и как набор примеров с пояснениями):



Видно, что компоненты могут быть вложенными друг в друга, что они могут иметь интерфейсы (так же, как и классы, но на диаграммах компонентов чаще всего используется только “леденцовая” нотация). У компонентов есть порты, которые могут предоставлять или потреблять интерфейсы, но порты часто не рисуются, а подразумеваются, поскольку обычно порт имеет только один интерфейс. Порты бывают полезны для изображения delegирования — что компонент просто перенаправляет запросы вложенному компоненту. Слова <> и “internal structure” опциональны (и обычно не пишутся).

# Лекция 4: Моделирование и анализ

Юрий Литвинов

yurii.litvinov@gmail.com

## 1. Введение

В этой лекции речь пойдёт не про UML (точнее, не только про UML), а про использование моделирования в фазе анализа предметной области. Анализ требований, анализ предметной области — самые первые и самые важные этапы проекта, поэтому и моделирование на этих этапах применяется очень активно, даже теми, кто ненавидит UML всей душой. Существует довольно много разных визуальных языков, краткий обзор их и будет в этой лекции.

## 2. CASE-системы

Начнём мы с того, что, наверное, следовало бы рассказать ещё в самом начале — про инструментальные средства, которые позволяют, собственно, пользоваться визуальными языками и моделировать. Эти инструментальные средства называются CASE-системами (Computer-Aided Software Engineering), или более специфично — UML Tool и т.п. для не-UML инструментов.

Сам термин CASE кажется слишком общим — “разработка ПО, поддержанная компьютером”, казалось бы, это не имеет отношения к моделированию особо. И правда, термин “CASE” появился в 80-е годы, когда разработка ПО для компьютера *на* компьютере была новой свежей инженерной идеей. CASE-системами тогда называли всё, что помогает разрабатывать ПО, включая IDE и даже текстовые редакторы. При этом в те времена системы программирования считалось немодным делать минималистичными, как принято сейчас в некоторых сообществах — блокнотик и набор консольных тулов — обычные системы программирования 80-х представляли интерфейс для работы с компилятором и отладчиком, средства написания документации, рисования диаграмм и вообще старались поддерживать весь цикл разработки. Постепенно произошла специализация средств разработки, так что CASE-системами стали называть исключительно инструменты для работы с визуальными языками. Впрочем, профессиональные среды разработки от поддержки UML никогда не отказывались, и, например, Visual Studio всегда умела рисовать что-то, отдалённо похожее на диаграммы классов UML, и в последнее время только наращивает поддержку UML-диаграмм. Ещё, кстати, чтобы всех запутать, в конце 90-х появились так называемые “визуальные” среды разработки (названия “Visual Studio”, “Visual Basic”, например, наследие тех времён), они к визуальным языкам никакого отношения не имеют — визуальность сводится к наличию редактора форм прямо в среде.

Стоит обратить внимание, что CASE-системы — это не графические редакторы. Например, редактор Inkscape содержит всё необходимое для рисования UML-диаграмм — прямоугольники, линии, текст, возможность редактировать форму фигур после их рисования (поскольку редактор векторный). Тем не менее, это не CASE-система, да и рисовать UML-диаграммы в Inkscape — это плохая идея. Отличие в том, что CASE-системы до какой-то степени “понимают”, что в них рисуют: они знают, что класс — это класс, у него есть имя, поля и методы (а не просто текст на фигуре), они знают синтаксис объявления полей и методов, они в целом знают синтаксис диаграмм и будут ругаться, если вы делаете что-то противоречивое. Они часто имеют некоторый программный интерфейс, позволяющий третьесторонним инструментам получать доступ к информации в модели. Все CASE-системы разные, поэтому степень “понимания” у них может быть разной, от наличия графического примитива “класс” в палитре у самых простых рисовалок до автоматического поиска антипаттернов проектирования у самых продвинутых. Хороший критерий, позволяющий отличить CASE-систему от просто графического редактора — наличие возможности генерации кода по диаграмме.

## 2.1. Типичная функциональность CASE-инструментов

Перечислим, чего можно ожидать от среды визуального моделирования.

- Набор визуальных редакторов — чтобы рисовать диаграммы. Причём, редакторов, как правило, несколько (например, если инструмент поддерживает UML, то имеют свой редактор как минимум самые популярные диаграммы UML), часто поддерживается несколько разных визуальных языков (UML, ER, BPMN и т.д.).
- Репозиторий — централизованное хранилище информации о моделях. Обеспечивает связь (и непротиворечивость) разных представлений одной и той же сущности. Как правило, имеет программный интерфейс, позволяющий третьесторонним инструментам получать из него осмысленную информацию.
- Набор генераторов — они генерируют код по моделям. Как правило, не исполнимый (часто CASE-системы ограничиваются заглушками для классов, единицы умеют ещё генерить код методов по диаграмме автоматов). Как правило, генераторов много, в разные языки программирования.
- Текстовый редактор — для редактирования сгенерированного кода или документации. Часто поддерживает всякое продвинутое форматирование.
- Редактор форм — далеко не во всех CASE-системах такое бывает, но иногда встречается. Наследие времён, когда CASE-система была основным инструментом программирования и должна была покрывать все этапы жизненного цикла.
- Средства обратного проектирования (reverse engineering) — парсеры, которые могут преобразовывать код на разных языках программирования (или байт-коды) в диаграммы. Как правило, диаграммы классов. Многие CASE-системы пытаются обеспечить двухстороннюю связь между диаграммами и кодом (по диаграммам генерировать код, вносить в него изменения руками и видеть их на диаграммах), у совсем немногих получается делать это хорошо. Например, Visual Studio так умеет, но как

раз потому, что она не CASE-система, а среда разработки — она внутри хранит код в распарсеннем состоянии, а модели и код в редакторе по сути его просто отображают.

- Средства верификации и анализа моделей — большая редкость в существующих CASE-системах, однако какие-нибудь Ultimate-версии лидеров рынка могут предложить довольно продвинутые возможности в этом плане.
- Средства эмуляции и отладки — тоже редкость. Запускать и отлаживать можно диаграммы автоматов, иногда диаграммы активностей, но тоже скорее в Ultimate-версиях нескольких инструментов.
- Средства обеспечения командной разработки — диаграммы надо версионировать и выкладывать в общий доступ, но git или svn тут не подходят, потому что они не могут адекватно показывать различия (и вообще сравнивать диаграммы). Любой уважающий себя CASE-инструмент имеет своё решение этой проблемы, будь то сетевой репозиторий или даже настоящая система контроля версий для диаграмм. Все они, конечно, не могут взаимодействовать друг с другом. Хотя диаграммы можно выкладывать и в обычные системы контроля версий типа git как бинарные файлы (без всякого сравнения, разграничения доступа, совместного редактирования и прочих благ), поддержка на таком уровне тоже иногда есть.
- API для интеграции с другими инструментами — это могут быть самописные генераторы, системы сборки, системы генерации документации, отчётов и т.д. и т.п. API обычно предоставляется на уровне репозитория, внешние инструменты могут запрашивать элементы диаграмм и их свойства.
- Библиотеки шаблонов и примеров — подавляющее большинство CASE-систем предлагают набор примеров или шаблонов, с которых можно начать проектирование. Удобно для новичков и экономит время профессионалам.

## 2.2. Примеры CASE-инструментов

Существующие CASE-инструменты можно условно разделить три категории.

- “рисовалки” — не очень умные инструменты, которые позволяют удобно рисовать диаграммы и иногда немного генерить по ним код, но не пытаются помогать с архитектурой или отладкой программы. Используются прежде всего как графические редакторы, специально заточенные под рисование диаграмм. Иногда люди увлекаются и начинают хотеть от них большего (например, генерации исполнимого кода по модели в Visio), но для этого есть лучшие альтернативы. Примеры таких инструментов:

- Microsoft Visio — часть пакета Microsoft Office, на самом деле редактор диаграмм вообще, UML там один из десятков разных вариантов (от диаграмм из кружочков и стрелочек до планов помещений). Причём, UML, хоть и есть в стандартной поставке, там не очень продвинутый (некоторых элементов нотации не хватает), так что лучше отдельно поставить плагин с полноценной поддержкой UML (благо в Visio есть развитая pluginная система). Visio очень

популярен в бизнес-среде, но платный (и, кажется, даже не входит в студенческую подписку СПбГУ, хотя и есть в компьютерных классах), и работает только под Windows.

- Dia — Visio для Linux. Как часто бывает в Linux, бесплатна, с открытым исходным кодом, есть в репозитории любого уважающего себя дистрибутива, имеет кучу плагинов (в том числе, поддержку UML), умеет генерировать код. Больше практически ничего не умеет, поэтому как настоящая CASE-система не используется.
  - SmartDraw — рисовалка диаграмм вообще, не только программистских. Работает под виндой и имеет веб-версию, но платная.
  - LucidChart — примерно то же самое, несколько менее платное в том смысле, что сколько-то простых диаграмм на одного пользователя можно рисовать бесплатно. Имеет только веб-версию (и вроде как мобильные версии) и очень агрессивную рекламу.
  - Creately — простая, но относительно удобная веб-рисовалка. Рисует страшные как моя жизнь диаграммы, но если надо быстро что-то нарисовать без установки и длительного процесса регистрации, Creately вполне подойдёт.
- Полноценные CASE-системы — то самое, про что шла речь в предыдущем разделе. Как правило, платные, но, как правило, имеются бесплатные Community-версии, поэтому рекомендую пользоваться именно этими штуками, а не “рисовалками”. Как правило, все такие штуки десктопные, кроссплатформенные, с несколько урезанной браузерной версией. Популярные примеры:
- Enterprise Architect — довольно популярный в IT-индустрии инструмент, более-менее всё умеет (не только UML, но и BPMN, SysML и другие полезные штуки) и не очень дорог (от 300\$ за лицензию и без бесплатной версии, так что для бедных студентов или инди-разработчиков так себе, но даже для средних стартапов это копейки).
  - Rational Software Architect — бывший Rational Rose (самый первый инструмент, поддерживавший UML), переписанный на платформе Eclipse. Тоже более-менее всё умеет, зато дороговат и опять-таки без бесплатной версии. Ни разу не видел, чтобы его использовали при практической разработке ПО, возможно это как-то связано с крайним неудобством его официального сайта. Достоин упоминания из-за отличной поддержки архитектурных рефакторингов.
  - MagicDraw — говорят, довольно хорошая и довольно популярная CASE-система, но, опять-таки, не видел её в деле.
  - Visual Paradigm — сам ей пользуюсь и встречал в индустрии, очень рекомендую. Умеет очень много чего, но основное её достоинство — это usability. И наличие Community-версии.
  - GenMyModel — скорее, очень продвинутая браузерная “рисовалка”, чем настоящая CASE-система, но умеет генерировать код, умеет UML, BPMN и ещё некоторые нотации, имеет репозиторий, так что попала именно в эту категорию.

В отличие от перечисленных выше настоящих CASE-систем, вообще не имеет десктопной версии, зато бесплатна для личного использования. Рекомендую как продвинутую замену Creately.

- Прочие инструменты. Направлены в основном на быстрое иллюстрирование документации или веб-страниц чем-то, похожим на UML-диаграммы, позволяют текстом описать, что надо нарисовать. Наверное, будет приятно хардкорным кодерам, которые мышку в руках никогда не держали. Примеры (рекомендую покликать на ссылки, чтобы хотя бы знать, что так бывает):
  - <https://www.websequencediagrams.com/> — как намекает название, инструмент для рисования диаграмм последовательностей UML. Диаграммы описываются на очень простом текстовом языке, например, A->B: text позволит нарисовать диаграмму с двумя объектами, один из которых шлёт другому сообщение «text».
  - <http://yuml.me/> — генерирует по параметрам в URL картинки, которые можно вставлять на любую HTML-страницу. Например,  вставит диаграмму классов с двумя классами и ассоциацией между ними.
  - <http://plantuml.com/> — генерирует картинки (и даже ASCII-арт) по текстовому описанию. Например, Class01 <|-- Class02 сгенерирует диаграмму классов с двумя классами, один наследник другого. Не так удобно эти картинки куда-либо встраивать, зато может рисовать практически любые UML-диаграммы, и даже очень сложные, с десятками классов и кучей разных связей.

### 3. Моделирование требований

Теперь вернёмся к разговору об использовании визуальных моделях при разработке программного обеспечения. Первый этап жизненного цикла любого программного продукта — это сбор и анализ требований, и уже на этом этапе применяются техники, связанные с визуальными языками. Для чего вообще нужно формализовывать требования:

- Прежде всего, для того, чтобы понимать, что мы хотим сделать. Требования нужны самим разработчикам (есть известная поговорка “Без хорошего ТЗ получается... не очень”).
- Формализованные требования — это своего рода соглашение между разработчиками, заказчиками и пользователями. Обратите внимание, что заказчики и пользователи часто не одно и то же лицо, и требования у них могут быть разными. Например, при разработке компьютерной игры заказчиком выступает издатель и его цель — заработать как можно больше денег, а пользователями выступают, например, ленивые школьники, цель которых — весело провести время. Фиксация требований позволит их, во-первых, оценить на предмет соответствия ожиданиям всех участников процесса, во-вторых, обсудить со всеми участниками, чтобы потом ни к кому ни у кого не было претензий.

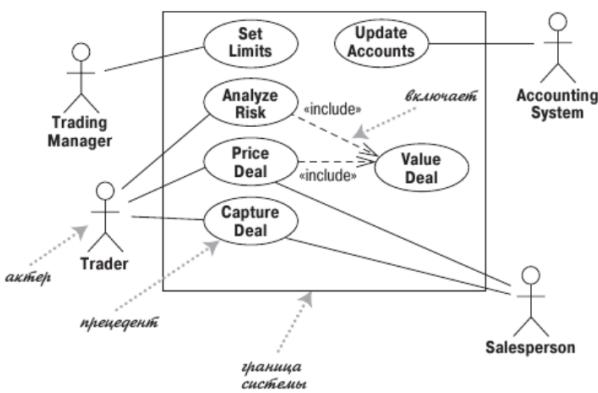
- Требования — это чёткое обозначение границ системы: что мы хотим делать, а что точно не хотим. Это позволяет противостоять feature creep-у, эффекту постепенного появления новых требований в процессе разработки, который может сильно затянуть или даже сорвать проект.
- Требования — это входные данные для следующей фазы жизненного цикла, планирования проекта.

Чаще всего требования описываются словесно, неформально или полуформально (например, через “user story” в Scrum и других Agile-методологиях). Есть полуформальные визуальные языки описания требований (например, диаграмма случаев использования UML, но она требует и текстовых описаний тоже), есть более-менее формальные модели, например, диаграмма требований SysML.

### 3.1. Диаграмма случаев использования UML

Первый визуальный язык, используемый для анализа требований (он же самый популярный), который мы рассмотрим — диаграммы случаев использования UML (также известные как диаграммы прецедентов, use case-diаграммы). Они появились до UML, как часть методологии OOSE, разработанной Айваром Якобсоном в 1992 году. OOSE на самом деле строилась вокруг случаев использования, их предлагалось продумать первыми, затем на их базе уже проектировать остальную архитектуру, непрерывно поддерживая связь со случаями использования (чтобы не делать лишней работы и быть уверенными, что вся требуемая функциональность где-то реализована). Потом Якобсон перешёл на работу в Rational и стал одним из трёх основателей UML, диаграммы случаев использования попали в язык практически без изменений.

Диаграммы очень простые, состоят всего из двух типов сущностей и границы системы:



© М. Фаулер, UML. Основы

- Акторы (или актёры, роли, это всё неудачные переводы со шведского на английский и с английского на русский) — внешние сущности, использующие систему. Это могут быть группы пользователей, объединённые общей целью использования системы (роли, собственно). Это могут быть также внешние программные системы, которые

пользуются нашей. Иногда в качестве акторов рисуются и другие программные системы (или люди), которые нужны нашей системе для работы, но это неканонично и лучше так не делать, чтобы не путать читателей.

- Случаи использования (прецеденты) — цель использования системы актором. Это одно-два-три слова, описывающие, что пользователь хочет в итоге получить (например, «Проанализировать риски», «Оценить сделку»). Их часто путают с функциональностью системы, например, «Залогиниться» — это не то же самое. Пользователь никогда не имеет целью использования системы залогиниться, в лучшем случае он хочет безопасного выполнения всех операций, но нефункциональные требования — это тоже не случаи использования, так что на этой диаграмме не рисуются.

Случаи использования формулируются слишком кратко, чтобы быть полезными сами по себе, поэтому каждый случай использования должен раскрываться в один или несколько *сценариев использования* системы. Сценарии использования как раз и говорят, что и в каком порядке делает пользователь, чтобы достичь цели (например, логинится -> выбирает нужную сделку -> анализирует риски -> аписывает результаты анализа в соответствующую форму). Сценарии чаще всего описываются просто текстом, часто неформальным, иногда как-то структурированным (например, user story из Scrum). Если вы очень любите визуальное моделирование, сценарии использования можно описывать диаграммами активностей UML.

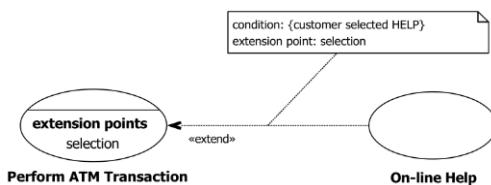
Случаи использования на диаграмме могут быть связаны отношениями, позволяющими их структурировать. Например, отношение *Include*:



© OMG, UML 2.5 Specification

*Include* означает, что один случай использования включает в себя другой, например, при снятии налички пользователь хочет, чтобы система идентифицировала его банковскую карту (вряд ли пользователь может отдельно хотеть именно этого, так что пример не очень каноничен, зато из стандарта языка).

Отношение *Extend*:

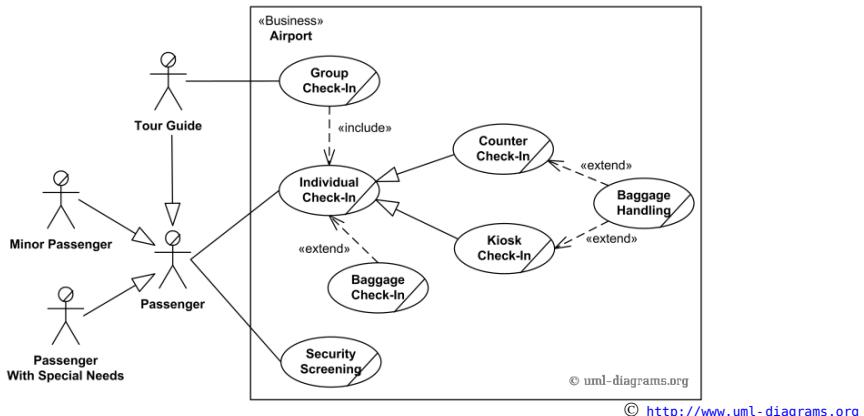


© OMG, UML 2.5 Specification

Это отношение означает, что один случай использования имеет одну или несколько точек расширения, в которых может использоваться функциональность, связанная с другим случаем использования. Например, «Проведение транзакции с помощью банкомата» имеет точки расширения, где пользователь может получить справку по использованию системы. Это может выражаться в связанных сценариях использования, например, словами

«А теперь пользователь хочет посмотреть баланс, но если он не знает, как это сделать, он может нажать на кнопку “показать справку”». Отношение «extend» можно понимать как уточнение отношения «include».

Вот пример побольше и посодержательней, регистрация в аэропорту:



Тут показано ещё и наследование — как между акторами, так и между случаями использования. Наследование, как обычно, является способом классификации сущностей — например, пассажир бывает малолетним, инвалидом или экскурсоводом. И если пассажирам вообще доступна индивидуальная регистрация и досмотр (вряд ли досмотр можно назвать целью пассажиров, но всё же), то экскурсоводу помимо всего этого доступна ещё и групповая регистрация. То же касается случаев использования — регистрация бывает у стойки, а бывает через терминал. В любом случае, если у пассажира есть багаж, его надо сдать, и на то есть свой случай использования, являющийся расширением первых двух. А групповая регистрация состоит из набора индивидуальных регистраций, поэтому отношение «include».

### 3.1.1. Сценарии использования

Каждый случай использования раскрывается в один или несколько сценариев использования. Иногда используются формальные описания сценариев использования, рассмотрим их, чтобы понимать, что вообще там надо писать.

Типичная структура сценария использования такова.

- Заголовок — цель основного актора, собственно, случай использования.
- Заинтересованные лица, акторы, основной актор — сюда выписываются все роли, участвующие в сценарии, среди них выделяется одна — которая инициирует исполнение сценария.
- Предусловия — какие логические условия должны быть истинны, чтобы сценарий в принципе мог быть выполнен.
- Триггеры (активаторы) — что должно произойти, чтобы сценарий начал выполняться, при условии, что все предусловия выполнены.

- Основной порядок событий — что происходит, если всё идёт по плану. Сюда пишется наиболее типичная последовательность действий.
- Альтернативные пути и расширения — сюда пишется, что делать в нетипичной ситуации, тут же описываются точки расширения и реакция на ошибки.
- Постусловия — какие логические условия должны быть истинны после исполнения сценария.

Вот пример карточки сценария использования, сделанной по такой структуре (из книги R.M. Roth et al., System Analysis and Design):

<b>Use Case Name:</b> Request a chemical	<b>ID:</b> UC-2	<b>Priority:</b> High
<b>Actor:</b> Lawn Chemical Applicator (LCA)		
<b>Description:</b> The Lawn Chemical Applicator (LCA) specifies the lawn chemical needed for a job by entering its name or ID number. The system satisfies the request by reserving the quantity requested or the quantity available and notifying the Chemical Supply Warehouse of the pick-up.		
<b>Trigger:</b> A Lawn Chemical Applicator (LCA) needs a chemical for a job.		
<b>Type:</b> <input checked="" type="checkbox"/> External <input type="checkbox"/> Temporal		
<b>Preconditions:</b>		
1. The LCA identity is authenticated. 2. The LCA has necessary training and credentials on file. 3. The Chemical Supply datastore is up-to-date and on-line.		
<b>Normal Course:</b>		
1.0 Request a lawn chemical from the chemical supply warehouse. 1. The LCA specifies a chemical needed and the quantity needed 2. The system lists chemical and quantity on hand from Chemical Supply datastore a. If the quantity on hand is less than the quantity needed, the LCA specifies the quantity he will take b. Purchasing is notified of chemical shortage 3. The system gives the LCA a Chemical Pick-up Authorization for the quantity requested 4. The system notifies the Chemical Supply Warehouse of the chemical pick-up 5. The system stores the Lawn Chemical Request in the Chemical Request datastore		
<b>Postconditions:</b>		
1. The Lawn Chemical Request is stored in the Chemical Management System. 2. The Chemical Pick-up Authorization is produced for the LCA. 3. The Chemical Supply Warehouse is notified of the chemical pick-up. 4. Purchasing is notified of chemical outage.		
<b>Exceptions:</b>		
E1: Chemical is no longer approved for use (occurs at step 1) 1. The system displays message, "That chemical is no longer approved for use" 2. The system asks the LCA if he wants to request another chemical or to exit 3a. The LCA asks to request another chemical 4a. The system starts Normal Course again 3b. The LCA asks to exit 4b. The system terminates the use case		

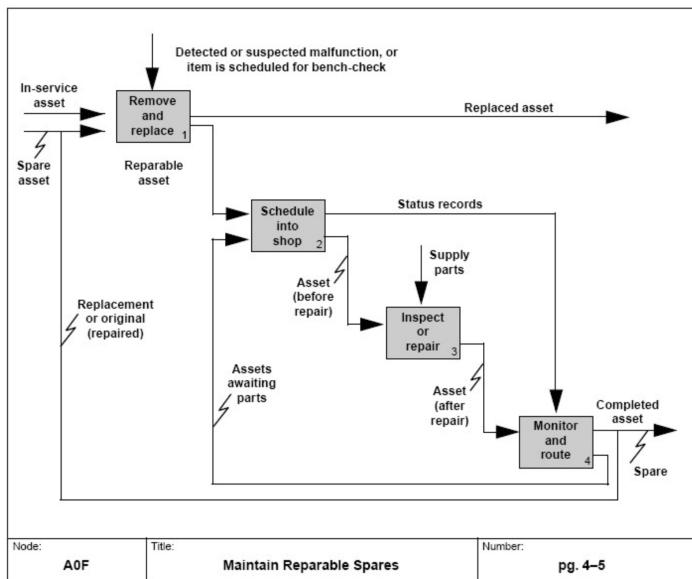
© R.M. Roth et al., System Analysis and Design

### 3.2. Контекстная диаграмма IDEF0

Диаграммы случаев использования хороши тем, что они очень просты и ни к чему не обязывают (из всего UML можно использовать только их, и они легко вписываются в любую методологию), так что применяются даже в Agile-командах иногда. Но есть ещё более простая нотация (и гораздо более древняя): контекстные диаграммы языка IDEF0.

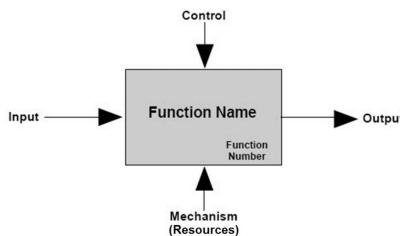
Вообще языки семейства IDEF (Integration DEFinition, в оригинале — ICAM DEFinition) разрабатывались аж в 1970-х годах по заказу минобороны США, когда у тех возникла потребность в разработке сложного mission-critical программного обеспечения и подхода “фигачить код” стало не хватать. Семейство на данный момент включает в себя 14 языков (как UML), но широкую известность получили только IDEF0 и IDEF1x, так что упоминаются в этом курсе только они.

Нотация IDEF0 (так же известная как SADT, Structured Analysis and Design Technique, хотя это не совсем верно, SADT появился раньше) используется для структурной декомпозиции системы (или бизнес-процесса, в который система встраивается). Процесс представляется в виде набора шагов, соединённых входами и выходами друг с другом. Диаграммы выглядят примерно вот так:



© <https://en.wikipedia.org/wiki/IDEF0>

Выходов у каждого блока всего один тип, хотя и может быть много, а вот входов три разных типа:



© <https://en.wikipedia.org/wiki/IDEF0>

Слева — входы блока, материалы, которые он перерабатывает в то, что получается на выходе. Сверху — управление, то есть то, что определяет, как блок работает (например, настройки или должностные инструкции). Снизу — механизмы, то есть то, что позволяет блоку работать (например, база данных или станки на конвейере). Опять-таки, входов каждого типа может быть много.

Каждый блок может быть далее раскрыт своей IDEF0-диаграммой, при этом все входы на исходной диаграмме должны быть входами и на диаграмме, раскрывающей блок, аналогично с выходами — каждую связь должно быть можно отследить.

В корне иерархии диаграмм лежит та самая контекстная диаграмма IDEF0, которая нам и интересна в контексте работы с требованиями. На ней рисуется всего один блок, означающий вообще всю систему. Входящими стрелками показываются входы, механизмы и управление для системы, приходящие извне (именно извне, всякие внутренние дела системы на этой диаграмме не рисуются), выходами — то, что полезного получается в результате работы системы. Небольшой пример (не из мира ПО, правда, но IDEF0 вообще чаще используются для моделирования бизнес-процессов и производственных процессов, чем для моделирования ПО, UML же есть — однако именно модели бизнес-процессов и нужны для того, чтобы грамотно описать требования):

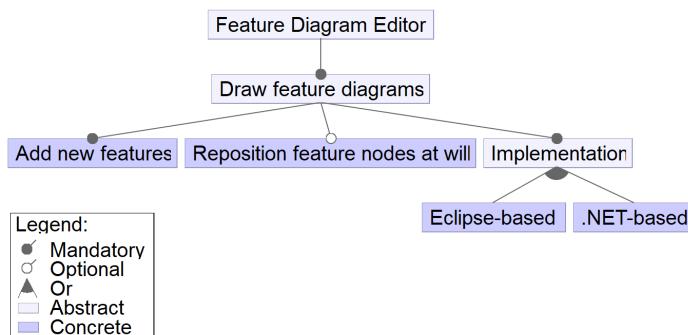


© <https://en.wikipedia.org/wiki/IDEF0>

Такая диаграмма кажется настолько простой, что полезность её сомнительна, но именно в этом её преимущества — она позволяет чётко определить границу системы, её входы и выходы, так, чтобы это помещалось на одном экране.

### 3.3. Диаграмма характеристик

Контекст, в котором система работает — это хорошо, но этого недостаточно, чтобы начать её реализовывать, нужны конкретные требования. Требования обычно иерархичны, поэтому и представлять их удобнее всего в виде дерева. Чаще всего это делают текстом, в виде списков, но есть и визуальная нотация: диаграмма характеристик (feature diagram). Выглядит она как-то так:

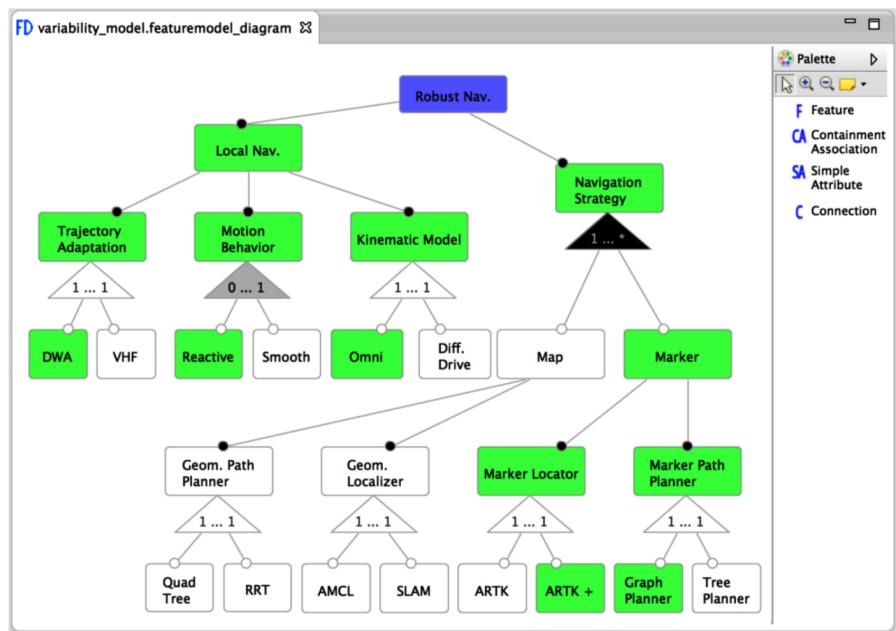


Нотация очень простая — характеристики бывают абстрактными (то есть теми, которые надо ещё декомпозировать перед тем как реализовать) и конкретными (пригодными к реализации), отношения между характеристиками бывают “или” (включающее и исключающее), обязательность и необязательность (то есть, без необязательной характеристики продукт вполне может существовать).

Для “повседневного” анализа требований диаграммы характеристик не очень удобны (отчасти потому, что их почти никто из CASE-систем не поддерживает, отчасти потому, что они получаются зело большими). Однако они активно используются в разработке линеек программных продуктов (например, если у вас есть Professional и Ultimate-версии, то можно отметить на общей диаграмме характеристик, какие характеристики куда попадают). Особенно хорошо это работает, если можно наладить автоматическую генерацию конфигурации продукта (или даже кода) по диаграммам.

Хороший пример такого подхода описан в статье D. Brugali et al., “Variability Modeling of Service Robots Experiences and Challenges” (2019) (точнее, у них про это целый проект BRICS, публикаций по нему много, эта даже не самая показательная, но самая свежая). Товарищи занимаются конфигурированием ПО для сервисных роботов, что особенно сложно, потому что роботы имеют разную аппаратную конфигурацию, работают в разном окружении и предназначены для разных задач. Поэтому и программные платформы, и конкретное ПО приходится выбирать из массы вариантов и адаптировать под конкретного робота, задачи и даже аппаратная конфигурация которого может меняться по ходу дела.

Общая идея такая: описать все возможные характеристики аппаратной платформы, характеристики доступного ПО и получить диаграммы характеристик в духе:

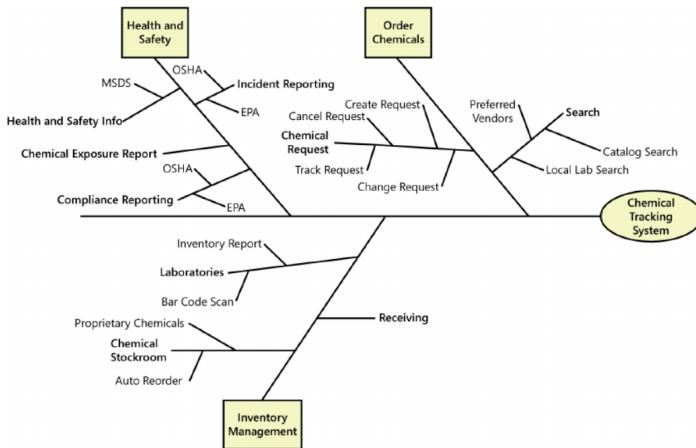


© D. Brugali et al.

Дальше под конкретную задачу можно просто выбирать нужные характеристики, а система сама сгенерирует код, который развернёт на роботе нужный софт и правильно его сконфигурирует. На самом деле, там не всё так просто, есть ещё отображения между характеристиками аппаратной платформы и характеристиками ПО, зависимости между характеристиками (например, если на роботе нет камеры, то распознавать маркеры на стенах не выйдет), есть типовая архитектура, которая, собственно, позволяет конфигурировать систему, но это уже детали реализации и не относится к теме лекции.

#### 4. Feature tree

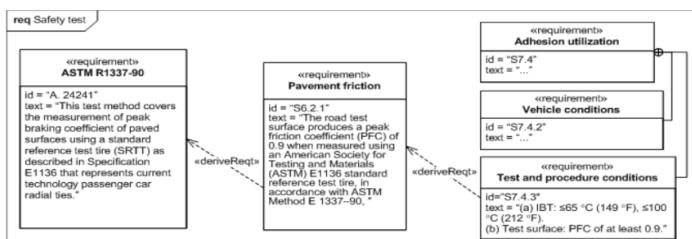
Диаграммы характеристик довольно громоздки и их сложно рисовать, поэтому во время первоначального анализа требований (например, brainstorm-а по поводу нового проекта) можно использовать похожую, но более простую нотацию: дерево характеристик (feature tree, или их ещё называют fishbone diagram из-за внешней схожести со скелетом рыбы). Выглядят они так:



“Голова рыбы” — это система в целом, от неё отходит “хребет”, от которого отходят основные фичи. Они дальше детализируются более мелкими фичами, рисуемыми как ответвления от основных. На такой диаграмме никаких взаимосвязей не показать, но как инструмент первоначального анализа она может быть очень полезна. А потом уже можно нарисовать случаи использования, диаграмму характеристик и т.д.

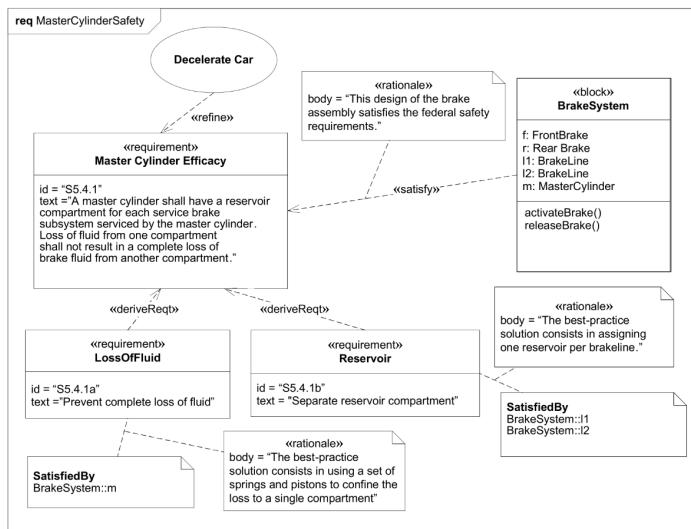
## 5. Диаграмма требований SysML

Есть более формальная нотация дерева требований, в языке SysML. SysML изначально создавался как профиль (т.е. стандартное расширение) языка UML для моделирования больших систем, включавших в себя как программные, так и аппаратные компоненты, а также людей и другие системы. Потом этот язык стал отдельным языком, туда добавили несколько новых видов диаграмм (которые будут упомянуты дальше в курсе), в частности, диаграмма требований. Выглядит она так:



Требования рисуются похоже на классы, со стереотипом `<<requirement>>`, именем и идентификатором требования, и, главное, текстовым описанием. Требования могут находиться в разных отношениях (например, `<<deriveReqt>>`, означающее, что одно требование является уточнением другого, или `satisfy`, означающее, что какой-то элемент системы, например, класс, удовлетворяет или реализует то или иное требование).

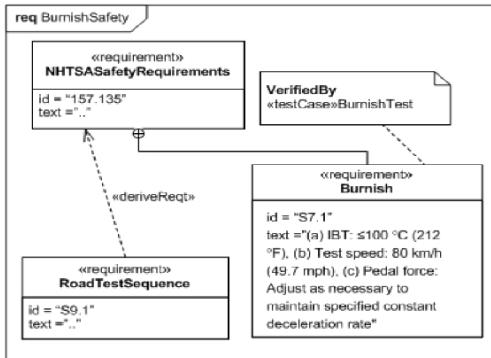
Вот более содержательный пример:



© OMG SysML 1.4 Specification

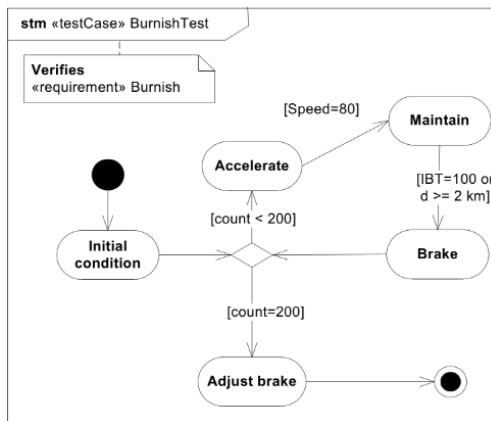
Как видим, на одной диаграмме могут рисоваться совершенно разные элементы. Случай использования «Decelerate Car» уточняется требованием «Master Cylinder Efficacy», которое реализуется подсистемой «BrakeSystem» (классы и целые подсистемы в SysML называются блоками). При этом «Master Cylinder Efficacy» уточняется наличием требования на невозможность полной потери тормозной жидкости и наличием разделенных резервуаров тормозной жидкости для каждой из линий тормозной системы. Каждое требование поясняется комментарием `<<rationale>>`, где написано, почему было принято такое решение.

Требования могут быть также уточнены сценарием тестирования (в SysML даже есть стандартные отношения `<<verifies>>/<<verifiedBy>>`). Сценарий тестирования может быть представлен в виде диаграммы активностей. Например, требования на работу тормозных колодок:



© OMG SysML 1.4 Specification

могут быть проверены следующим сценарием тестирования:



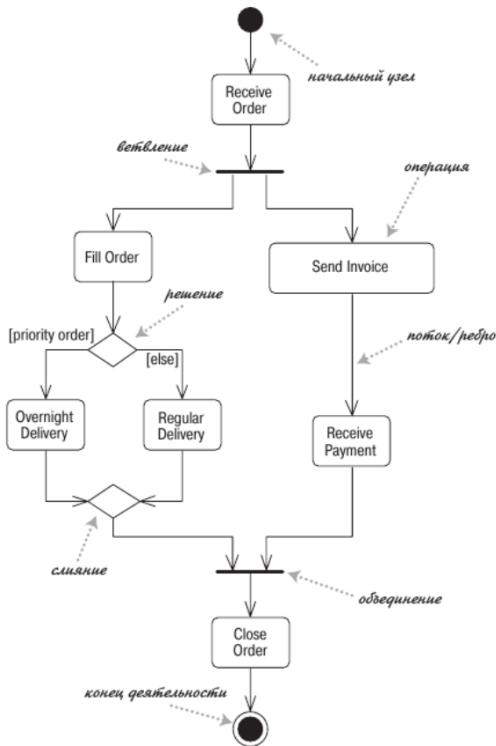
© OMG SysML 1.4 Specification

## 6. Диаграмма активностей UML

Важной задачей на этапе анализа, помимо сбора требований, является ещё и анализ бизнес-процессов, которые (или часть которых) мы хотим автоматизировать. Для этого применяются несколько нотаций — во-первых, диаграммы активностей UML, во-вторых, отдельный язык BPMN (Business Process Model and Notation), который, как и UML, состоит из нескольких видов диаграмм и специально предназначен для моделирования бизнес-процессов.

Но сначала про диаграммы активностей из UML, как более легковесный инструмент. Внешне они очень напоминают блок-схемы, но используются прежде всего не для моделирования алгоритмов (хотя и для этого тоже, иногда), а для моделирования поведения бизнес-процесса в целом. Кстати, термином “бизнес-процесс” в ИТ принято называть любой процесс работы в организации, может быть, и не связанный с бизнесом. Зачем вообще моделировать процессы — во-первых, для того, чтобы понять, как в существующий процесс вписывается разрабатываемая система, во-вторых, это хороший способ визуализации сценария использования системы, с точки зрения пользователя.

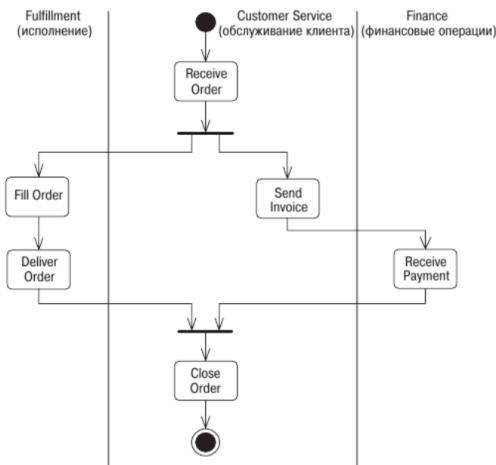
Кстати, это одна из двух диаграмм UML, для которой описана семантика её исполнения, на основе сетей Петри. Вторая диаграмма — это диаграмма конечных автоматов, про которую несколько попозже. Выглядит диаграмма активностей (также иногда встречается вариант перевода “диаграмма действий”) так:



© М. Фаулер, UML. Основы

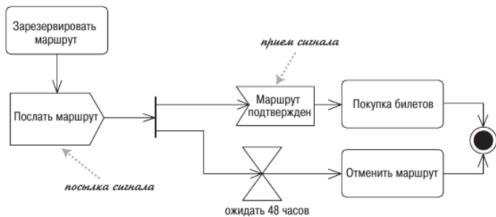
«Ветвление»/«объединение» работает как fork/join для параллельных потоков — «ветвление» разделяет исполнение на два или больше параллельных потоков, «объединение» ждёт, пока все входящие в него потоки закончат исполнение, и только после этого отдаёт управление дальше. Ничто не мешает потоки не объединять, тогда исполнение заканчивается, как только хотя бы один поток дойдёт до блока «конец деятельности». По синтаксису каждый блок «решение» может иметь несколько веток (в этом смысле он похож на оператор switch/case в текстовых языках), но все ветки должны сходиться на блоке «слияние». Причём, в отличие от блок-схем, условие пишется не в ромбике, а над каждой исходящей стрелкой, и условия должны быть взаимоисключающими.

Ещё на диаграмме активностей можно показать разделение работ по отделам организации или частям системы. Это важно в анализе бизнес-процессов, поскольку визуализирует разделение ответственности между частями организации. Рисуются разделы так:



© М. Фаулер, UML. Основы

А ещё, в отличие от блок-схем, есть возможность показать асинхронное взаимодействие с помощью сигналов. Рисуются они вот так:



© М. Фаулер, UML. Основы

Сигналом чаще всего является отправка документа, но может быть и сетевой запрос, и электронное письмо, и что-то ещё, что отправляется за границы моделируемого бизнес процесса. Блок «посылка сигнала» тут же возвращает управление и процесс продолжается вне зависимости от того, принял кто-то сигнал или нет. Блок «приём сигнала» ждёт указанный сигнал и продолжает процесс только когда сигнал поступит. Ещё есть блок «таймер», который можно понимать как отложенный сигнал самим себе — процесс ждёт указанное время, затем продолжается. На диаграмме выше показан типичный запрос с таймаутом.

## 7. Business Process Model and Notation

Для более продвинутого анализа бизнес-процессов используется отдельный язык BPMN (Business Process Model and Notation), которые не входит в UML, хотя и является его близким родственником. Диаграммы активностей хороши, если у вас один-два бизнес-процесса с тремя-четырьмя участниками, однако в реальной жизни нередки ситуации, когда у организации более сотни субподрядчиков, и взаимодействие с каждым — отдельный бизнес-процесс, а работа организации — это тоже бизнес-процесс, включающий в себя эти бизнес-процессы. Если вдруг не повезло автоматизировать бизнес-процессы такого масштаба, то без BPMN не обойтись. Правда, в этом случае не стоит доверять анализ бизнес-процессов архитектору, лучше нанять отдельного бизнес-аналитика. Поэтому

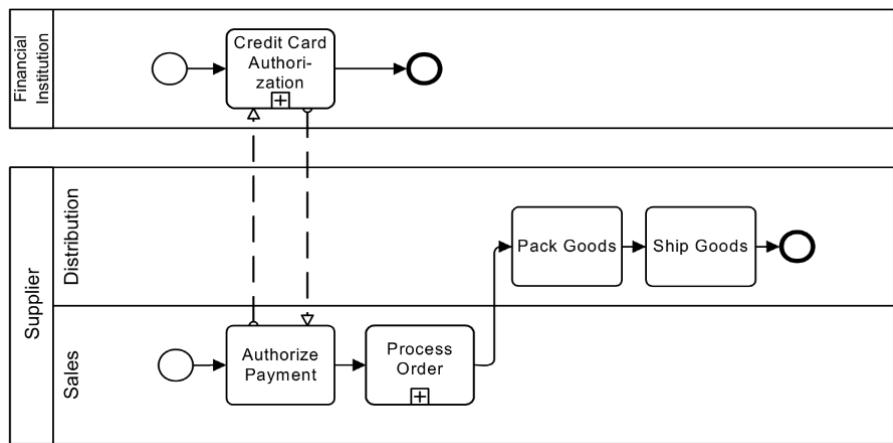
BPMN несколько реже встречается в программистской практике, чем диаграммы активностей UML, и поэтому в этом курсе про BPMN будет существенно меньше, чем могло бы быть, но поскольку архитектуру надо как-то общаться с аналитиками и вообще знать, что такое бывает, про BPMN всё-таки будет.

Стандарт BPMN 1.0 был принят в 2004 году, как раз примерно в те времена, что и UML 2.0, тем же консорциумом Object Management Group, который управляет UML. BPMN также определён с помощью метамодели и стандарты очень похожи по принципам определения языка, но BPMN, тем не менее, отдельный язык (точнее, как и UML, набор языков).

BPMN можно воспринимать как сильно продвинутые диаграммы активностей. Он позволяет, в частности, визуализировать взаимодействие нескольких процессов (в отличие от диаграмм активностей, где посылка и приём сигналов просто не связанные между собой блоки, в BPMN рисуются нормальные стрелочки), имеет гораздо больший набор блоков-ветвлений (в том числе и с параллельными потоками), блоков ожидания разных событий, качественную поддержку исключений. Так же, как и диаграммы активностей, BPMN имеет исполнимую семантику, но, в отличие от диаграмм активностей, эта семантика может быть использована для реального исполнения бизнес-процесса в информационной системе — благодаря стандартизованным правилам преобразования диаграмм на BPMN в XML-документы на BPEL (Business Process Execution Language), настоящем исполнимом языке описания бизнес-процессов, который поддерживает многие движки-исполнители<sup>1</sup>.

## 7.1. Диаграмма процессов

Вот так выглядит самая, пожалуй, известная диаграмма BPMN (которую часто путают с BPMN вообще, например, в википедии) — диаграмма процессов (process diagram):



Тут изображены два бизнес-процесса двух независимых организаций, общающихся посылкой и приёмом сообщений. Бизнес-процесс одной из организаций поделен на два раздела, но тем не менее, это один бизнес-процесс, на что указывает единый поток управления. Плюсик внутри активности означает, что активность раскрывается в другой, более дет-

<sup>1</sup> см. [https://en.wikipedia.org/wiki/List\\_of\\_BPEL\\_engines](https://en.wikipedia.org/wiki/List_of_BPEL_engines)

тализированный бизнес-процесс. Пользоваться такими диаграммами можно как диаграммами активностей UML, а подробнее прочитать про их синтаксис можно, как ни странно, в русской википедии (<https://ru.wikipedia.org/wiki/BPMN>), если не хотите ознакомиться с почти шестисотстраничным стандартом<sup>2</sup>. Вот табличка с видами событий оттуда:

	Начальные	Промежуточные	Завершающие
	Обработка	Генерация	
Простое			
Сообщение			
Таймер			
Ошибка			
Отмена			
Компенсация			
Условие			
Сигнал			
Составное			
Ссылка			
Останов			

© <https://ru.wikipedia.org/wiki/BPMN>

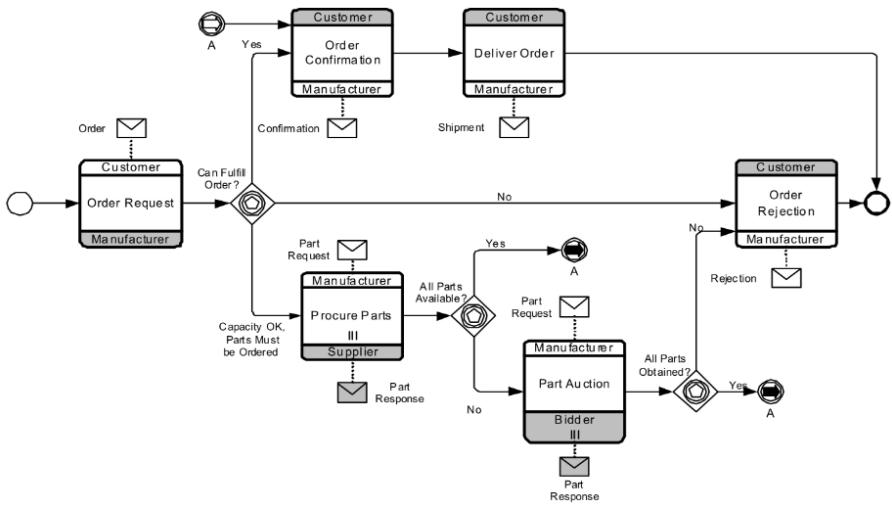
Пояснять тут её нет смысла, потому что в википедии всё подробно описано, но представление о том, что в BPMN бывает и почему BPMN лучше для анализа сложных процессов, чем диаграмма активностей UML, это даёт. А вот несколько менее эпичная таблица с типами логических операторов (опять же, из википедии, и, опять же, без пояснений):



© <https://ru.wikipedia.org/wiki/BPMN>

## 7.2. Диаграмма хореографии

Второй вид диаграмм BPMN — диаграмма хореографии (choreography diagram) — описывает исключительно взаимодействие между бизнес-процессами. На ней не рисуется кто что должен делать, на ней рисуется, кто когда с кем должен общаться. Выглядит это так:

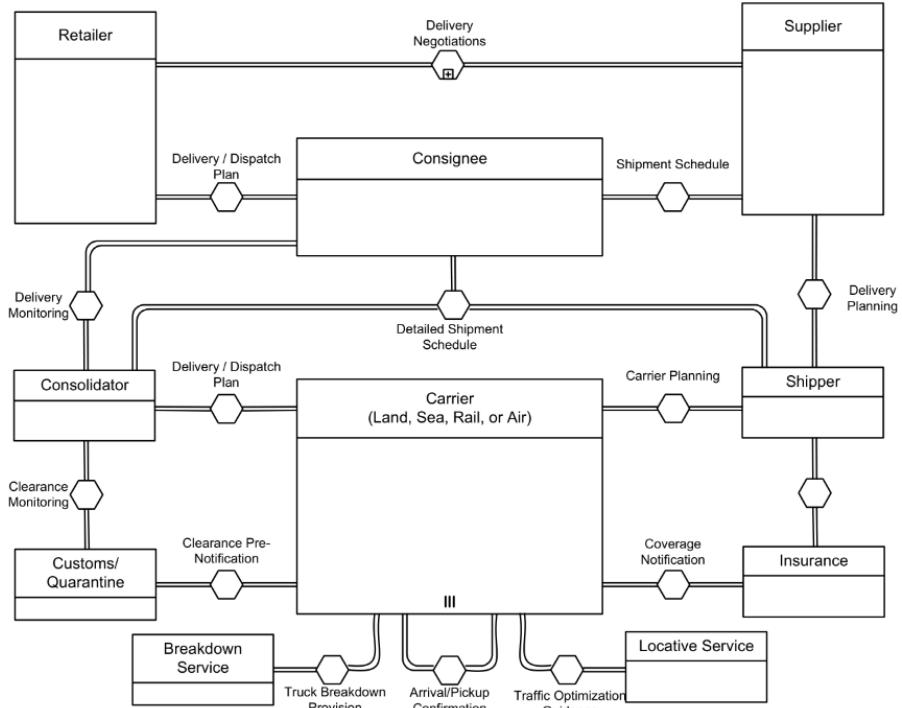


© OMG BPMN 2.0 Specification

Прямоугольники со скруглёнными углами — это точки общения между процессами, белым цветом выделен инициатор взаимодействия, серым — тот, кто должен ему ответить, конвертиком — сообщение или документ. Как видим, ветвления и события тут рисуются, а активности — нет. Такая диаграмма нужна, если попытка изобразить процессы в дорожках на диаграмме процессов привела бы к хаосу из стрелочек.

## 7.3. Диаграмма диалогов

И последняя диаграмма BPMN — диаграмма диалогов (conversation diagram), она показывает схему общения бизнес-процессов, не в смысле когда кто с кем общается, как диаграмма хореографии, а в смысле кто с кем в принципе может общаться. Выглядит диаграмма вот так:



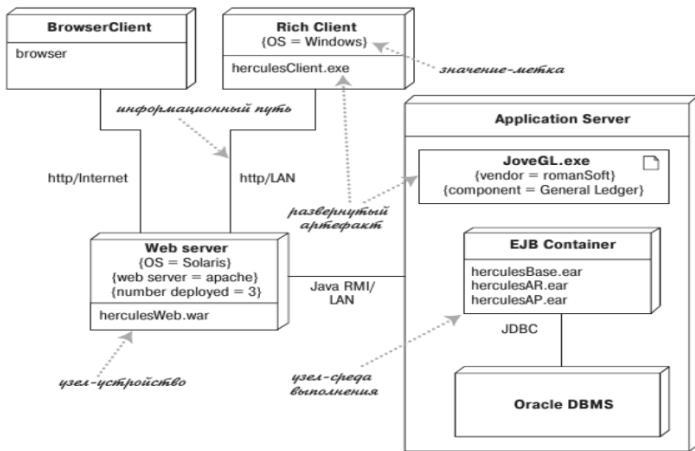
© OMG BPMN 2.0 Specification

Прямоугольниками показаны участники взаимодействия, двойными линиями с шестиугольниками — общение между участниками. Плюсик внутри шестиугольника показывает, что взаимодействий между участниками много и они могут быть уточнены диаграммами хореографий или диаграммами процессов без деталей (это когда просто рисуются дорожки и сообщения, без внутреннего содержимого, так тоже можно). Три вертикальные черты внутри участника взаимодействия означают, что участников на самом деле может быть много.

На этом закончим обзор BPMN (очень-очень краткий, там ещё куча разных тонкостей, про которые надо знать, чтобы стать нормальным аналитиком, но у нас тут курс по архитектуре). Можно дополнительно ознакомиться с примерами BPMN-диаграмм из небольшого, неформального, но очень полезного документа от OMG: <https://www.omg.org/cgi-bin/doc?dtc/10-06-02>

## 8. Диаграмма развёртывания UML

Диаграмма развёртывания UML (deployment diagram) используется для того, чтобы показать размещение логических элементов системы (компонентов, исполнимых файлов) на физических или виртуальных устройствах. Пример такой диаграммы:



© М. Фаулер, UML. Основы

Параллелепипедами на диаграмме рисуются устройства или похожие на устройства части системы (реальные компьютеры или группы компьютеров, базы данных, среды выполнения и т.д.). В них рисуются развёрнутые там артефакты (бинарники, компоненты и т.д.). Связи здесь возможны только между узлами (устройствами) и показывают один физические (или низкоуровневые логические) каналы связи между узлами, с указанием протокола взаимодействия. И узлы, и артефакты могут иметь дополнительные свойства, пишущиеся в фигурных скобках. Артефакты можно рисовать как подробно (прямоугольником), так и кратко (просто надписью с именем артефакта).

Казалось бы, развёртывание происходит ближе к концу разработки, так что не имеет отношения к анализу. Однако оказывается, что часто физические устройства оказываются известны заранее (например, заказчик уже купил два сервера и готов купить ещё пяток терминалов), и наличествующие физические устройства во многом определяют архитектуру системы. Тогда диаграмма развёртывания очень полезна как инструмент именно анализа — она позволяет отобразить уже существующую инфраструктуру и понять, как мы намерены её использовать.

Ещё одно полезное применение диаграммы развёртывания — обратное проектирование, когда система у заказчика уже есть, но никто не знает, как она устроена, а вам надо для неё чего-то написать. Как правило, даже если документация по коду совсем не сохранилась, у админов есть документация по сопровождению, или они просто знают, какую машину надо перезапустить, если повис такой-то сервис. Если отобразить эти знания на диаграмме развёртывания, получим первое приближение высокогоуровневой архитектуры системы.

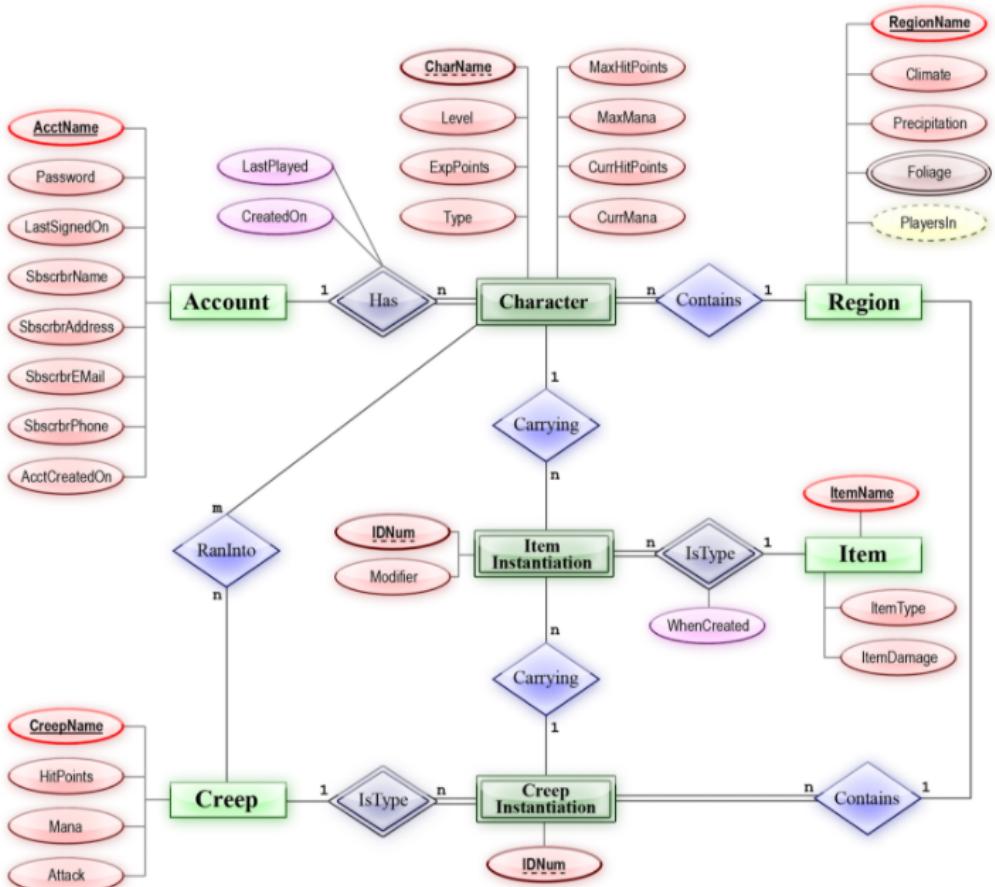
## 9. Диаграммы «Сущность-связь»

Следующая важная часть анализа предметной области — это анализ данных, с которыми предстоит работать нашей программе. Точнее даже поначалу анализ сущностей, наличествующих в предметной области, их взаимосвязей и свойств. Для этого тоже есть несколько визуальных языков, но, поскольку построению схем БД обычно хорошо учат на курсе по базам данных, тут только затронем этот вопрос, и то только его визуальную

составляющую. В данном случае ситуация несколько обратна BPMN — моделирование сущностей и связей предметной области и создание концептуальной схемы базы данных — это одна из основных деятельности архитектора, но раз она хорошо покрыта другими курсами, мы тоже только слегка её затронем.

Итак, для описания концептуальной модели предметной области чаще всего используются диаграммы «сущность-связь» (хотя могут и диаграммы классов UML). Для построения схемы реляционной базы данных используются практически исключительно диаграммы «сущность-связь», поскольку они семантически очень хорошо ложатся в реляционную модель данных. Хитрость в том, что у диаграмм «сущность-связь» есть несколько нотаций.

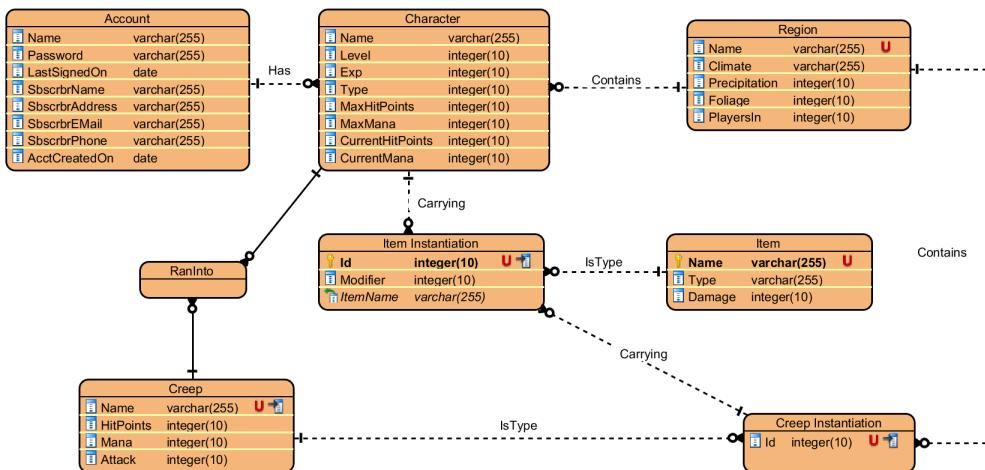
Первая нотация была предложена Питером Ченом ещё в 1976 году и выглядит так:



© <https://ru.wikipedia.org>

Прямоугольники означают сущности, овалы — их свойства, ромбы — связи между сущностями (которые сами могут иметь какие-то свойства). Связи могут иметь кратность (один к одному, один ко многим, даже многие ко многим), какой-то атрибут может быть назначен первичным ключом. Никогда не видел, чтобы эта нотация использовалась на практике.

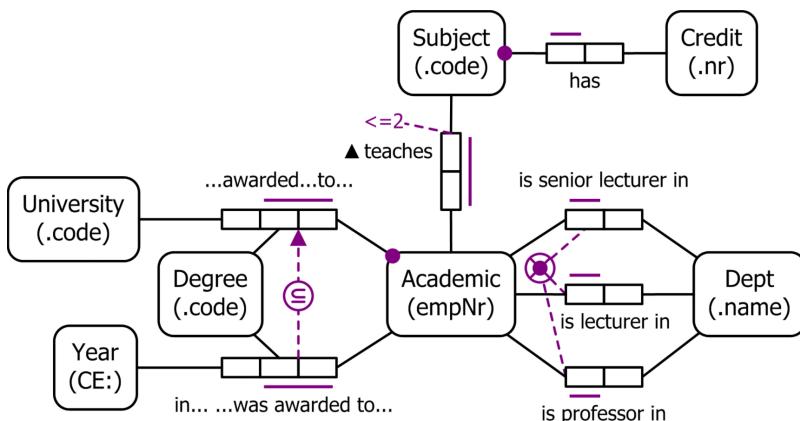
Реально используются сейчас разные варианты нотации «вороньей лапки»:



Нотация получила своё название за обозначение множественности связи (три разветвляющиеся линии на конце связи, похожие на птичью ногу). Её важное отличие от нотации Чена в том, что атрибуты в ней рисуются прямо внутри сущности, что делает диаграммы гораздо более компактными. Здесь тоже можно назначить один из атрибутов первичным ключом, задать ограничения уникальности и т.д., но детали нотации несколько различаются от инструмента к инструменту, поэтому не будем здесь на ней подробно останавливаться (благо такие диаграммы вы наверняка уже видели в контексте баз данных).

## 10. Object-Role Modeling

Object-Role Modeling (ORM) — это ещё одна нотация для построения концептуальной модели предметной области и, в конечном итоге, схемы базы данных проектируемой системы. Она гораздо реже используется, чем ER-диаграммы, но считается довольно прогрессивной и хорошо подходящей именно для анализа. Нотация очень проста, в ней есть только сущности и связи, атрибутов нет. Зато связи более хитрые, они часто п-арные (в ER-диаграммах были возможны только бинарные связи) и могут быть соединены ограничениями. Вот пример такой диаграммы, описывающей сотрудников некоего университета:



© <http://www.orm.net>

Сущность имеет схему идентификации (её единственный атрибут — признак, по которому её можно отличить от других), это то, что пишется в скобках. Связи играют роль таблиц в реляционной модели — каждая связь представляет собой кортеж из нескольких сущностей, сущности играют в связи *ролями*, обозначаемые прямоугольниками посреди связи. Над связью пишется вариант прочтения, например, «Университет такой-то присудил степень такую-то человеку такому-то». Это нужно прежде всего для лёгкости чтения диаграммы неспециалистами, и чтобы аналитик мог проверить естественным языком, что он делает что-то разумное. Горизонтальные и вертикальные полосы над ролями показывают, какие сочетания сущностей в данной связи должны быть уникальны. Роли могут быть связаны ограничениями, например, «исключающего или»: препод может быть либо преподавателем, либо старшим преподавателем, либо профессором на факультете.

Такие диаграммы используются реже ER-диаграмм в силу того, что для сколько-нибудь больших предметных областей ORM-диаграммы существенно больше по размеру, чем ER (каждый атрибут всё-таки надо моделировать как сущность). Зато такие диаграммы более устойчивы к изменениям в предметной области или нашего её понимания — если мы в ER-диаграмме что-то сделали атрибутом, а потом оказалось, что у этого чего-то тоже есть атрибуты, нам придётся полдиаграммы переделывать (потому что атрибут атрибуты иметь не может). В ORM же просто добавляем новую связь с новой сущностью, и всё. Поэтому ORM-диаграммы считаются удобнее на первых этапах анализа, когда наших знаний ещё недостаточно, чтобы понять, кто сущность, а кто атрибут.

Обратите внимание, что ORM в программировании чаще используется в другом смысле — Object-Relational Mapping. Это библиотеки (точнее, обычно целые технологии), которые представляют содержимое реляционной базы как набор настоящих объектно-ориентированных объектов и наоборот, так что при написании программы мы просто как обычно манипулируем объектами, меняем их свойства, создаём и удаляем и т.д., а ORM-система транслирует все наши действия в запросы к БД, беря на себя все проблемы с тем, что объектно-ориентированное программирование с реляционной моделью совсем не дружат. Например, в реляционной модели невозможно выразить наследование и даже отношение «многие-ко-многим», так что ORM-системы нетривиальны. Так вот, в этом разделе речь шла не про то.

# Лекция 5: Моделирование поведения

Юрий Литвинов

yurii.litvinov@gmail.com

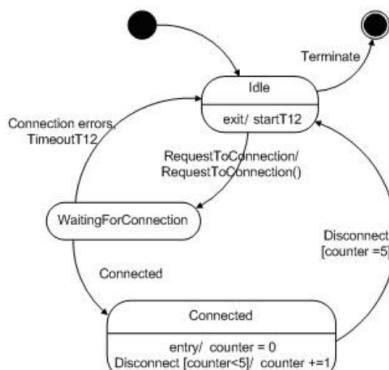
## 1. Введение

В этой лекции мы закончим обсуждение UML, рассмотрев диаграммы, которые используются на этапе разработки, или более конкретно, для моделирования поведения. Речь пойдёт не только про UML, но и некоторые другие формализмы, используемые для этой цели.

## 2. Диаграммы конечных автоматов

Диаграммы конечных автоматов (также известные как диаграммы состояний) — это на самом деле несколько упрощённые диаграммы Харела, предложенные им ещё в 1987 году, которые попали в UML с минимальными изменениями. Это второй вид диаграмм UML, имеющий исполнимую семантику. Предназначены эти диаграммы для моделирования поведения «реактивных» систем (или частей системы), то есть систем, которые находятся в некоторых чётко определённых состояниях, от которых зависит их поведение, и могут реагировать на события, переходя из состояния в состояние и, возможно, делая при переходах полезную работу. Примеры реактивных систем — это сетевое соединение (которое может быть открыто, закрыто, открываемо в данный момент, закрываемо, и в зависимости от этого передаёт или не передаёт пакеты), либо классический пример с торговым автоматом, с которого начался рассказ про моделирование вообще.

Выглядят диаграммы конечных автоматов так:

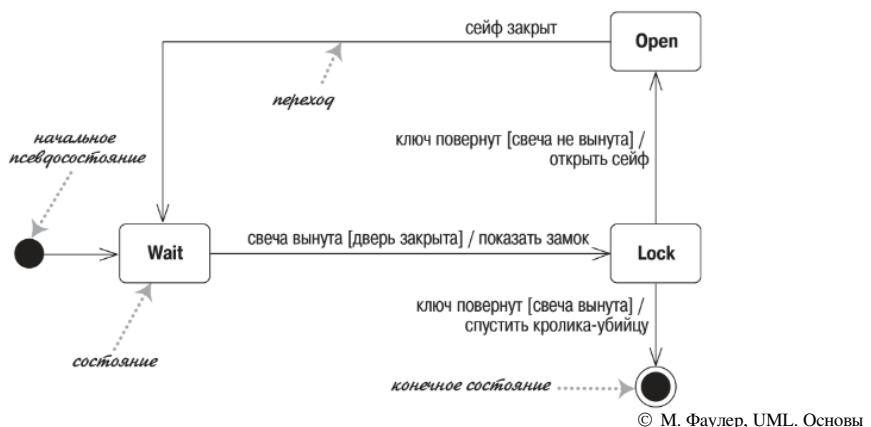


Прямоугольниками со скруглёнными углами рисуются состояния, у состояния есть имя и (опционально) действия, выполняемые в состоянии (например, действие по выходу или внутренний переход по событию, как у Connected — получив событие Disconnect, оно проверяет счётчик, и если счётчик меньше 5, он увеличивается на 1 и мы остаёмся в том же состоянии). Состояния связаны переходами, над переходом пишется событие, которое инициирует переход и, опционально, стражник (guard) (логическое условие, которое должно быть истинно, чтобы переход состоялся) и действие, выполняемое при переходе. События со стражниками должны быть взаимно исключающими, недетерминированные автоматы считаются некорректными. Есть псевдосостояния начала и конца, переход из псевдосостояния начала происходит мгновенно, переход в состояние конца заканчивает исполнение.

Внешне диаграммы конечных автоматов похожи на диаграммы активностей, но есть важные семантические различия:

- на диаграмме активностей рисуются активности, система в них не задерживается, а сразу переходит дальше; на диаграмме конечных автоматов рисуются состояния — стабильные отрезки жизненного цикла объекта, в которых он находится большую часть времени и может из них выйти только если что-то произойдёт;
- полезная работа на диаграммах активностей производится в активностях, на диаграммах автоматов — как правило, при переходе;
- диаграммы активностей моделируют один метод объекта (или какую-то функцию или что-то такое), диаграммы конечных автоматов — целый объект (состояния моделируются полями объекта).

Более подробно про синтаксис:



© М. Фаулер, UML. Основы

Внутри состояния могут быть:

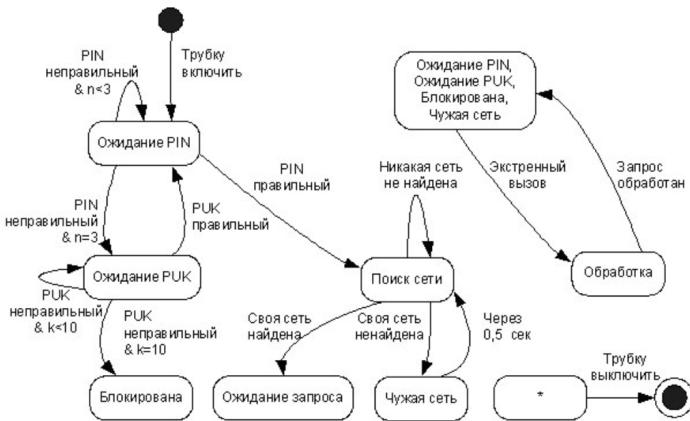
- entry activity — то, что делается при входе в состояние по любому из переходов;
- exit activity — то, что делается при выходе из состояния по любому исходящему переходу (и входная, и выходная деятельность — это, как правило, вызовы метода);

- do activity — деятельность, выполняющаяся всегда, когда система находится в таком-то состоянии (например, попытки подключения к сети для мобильного телефона);
- внутренний переход — переход по событию, который ведёт в то же состояние и не приводит к срабатыванию entry и exit activity. Переход вполне может быть полноценным переходом в то же состояние (рисуется как петля в графе), тогда entry и exit activity работают как обычно, хоть состояние и не меняется.

Событие, кстати, это нечто внешнее по отношению к системе, на что система может реагировать. Примеры событий — действие пользователя, сетевой пакет, считывание символа (если речь идёт об автоматном лексическом анализаторе, который, кстати, хоть и несколько необычный, но тоже пример реактивной системы, которая прекрасно моделируется конечными автоматами).

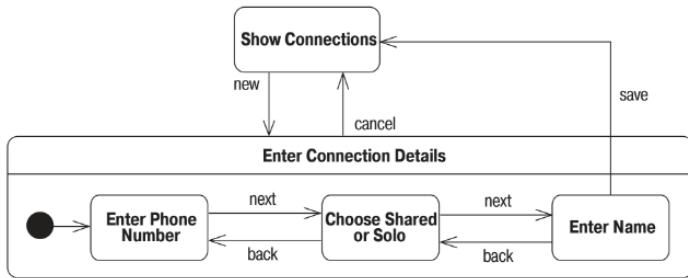
Надпись на переходе имеет следующий синтаксис:  
`[<trigger> [, , <trigger>]* '[' '<guard>' ]' ] ['/' <behavior-expression>]` — один переход может реагировать на несколько событий сразу, иметь optionalного стражника (в квадратных скобках) и через слеш действие (вызов метода или отсылку к диаграмме активностей, которая поясняет, что нужно делать при переходе).

Вот более содержательный пример автомата из работ Д.В. Кознова. Пример демонстрирует порядок работы мобильного телефона, начиная с включения и ввода PIN-кода и заканчивая подключением к сети:



Тут используется неканоничный синтаксис с псевдосостоянием “все состояния”, из которого ведёт переход в конечное псевдосостояние (чтобы не рисовать переход из каждого состояния в конечное) и используются не совсем каноничные надписи над переходами. Связано это с тем, что в те времена, когда на матмехе делались работы по диаграммам конечных автоматов, UML 2 ещё не было, а в UML первых версий синтаксис диаграмм конечных автоматов был менее проработан.

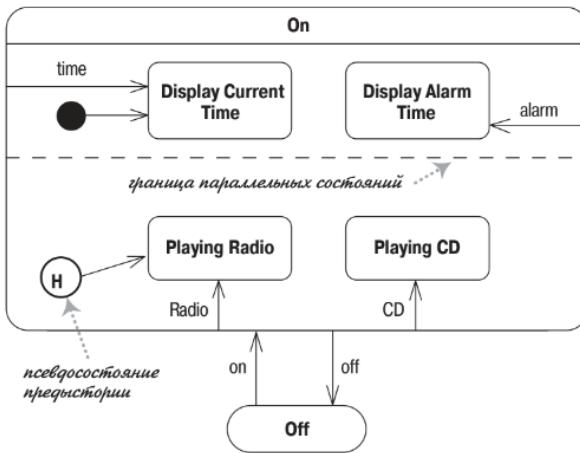
Более продвинутый синтаксис современных диаграмм конечных автоматов позволяет нарисовать пример выше более канонично. Есть вложенные состояния с переходами сразу из всех внутренних состояний:



© М. Фаулер, UML. Основы

Тут состояние «Enter connection details» содержит внутри свой конечный автомат, который начинает работать со стартового псевдосостояния когда выполняется переход «new». При этом переход «save» возможен только из состояния «Enter Name», а вот переход «cancel» возможен из любого вложенного состояния (это замена нестандартному псевдосостоянию со звёздочкой из диаграммы выше).

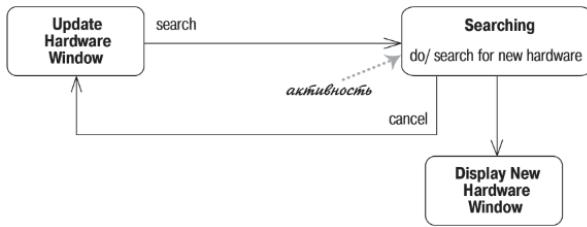
Ещё бывают параллельные состояния и псевдосостояние истории:



© М. Фаулер, UML. Основы

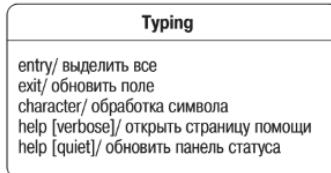
Это часы с радио и будильником. Проигрывание звука и время работают независимо, поэтому по сути это два автомата, работающих параллельно (что и показывает горизонтальная прерывистая линия, разделяющая параллельные подавтоматы). Стрелки от объемлющего состояния к вложенным означают, что система, находясь в объемлющем состоянии, реагирует на такие-то события и изменяет внутреннее состояние (например, часы по умолчанию показывают текущее время, но если пользователь нажал на кнопку «будильник», начинает показывать время, на которое будильник установлен). Псевдосостояние истории запоминает последнее вложенное состояние, в котором находился автомат, и возвращает автомат в него. Например, часы по умолчанию включают радио, но если пользователь включил воспроизведение компакт-дисков (если кто помнит, что это такое) и выключил часы, то при следующем включении они снова будут проигрывать компакт-диски.

А вот так рисуются активности внутри состояния:



© М. Фаулер, UML. Основы

А вот так — внутренние переходы и entry/exit-события, о которых шла речь выше:

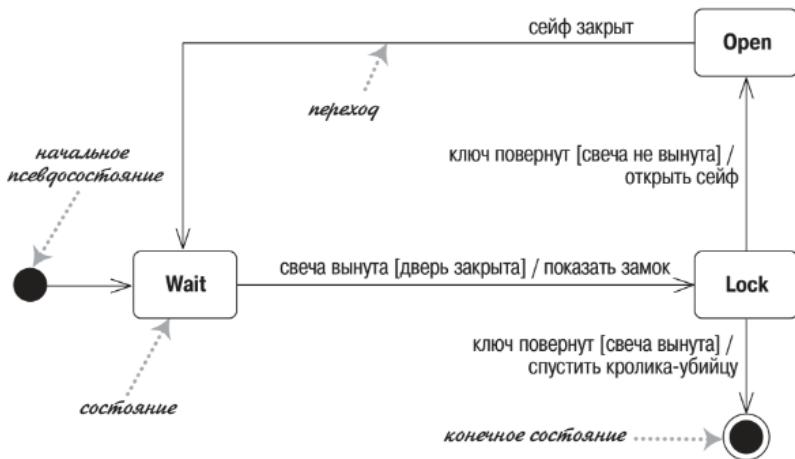


© М. Фаулер, UML. Основы

Как видим, синтаксис очень похож на то, что пишется при переходе.

## 2.1. Генерация кода

Конечные автоматы хороши тем, что по ним можно легко и приятно генерировать код, в силу их стандартизованной семантики. Рассмотрим наш пример с кроликом-убийцей:



© М. Фаулер, UML. Основы

Первый, самый простой способ сгенерировать код — это сгенерировать гигантский switch, внутри которого чуть менее гигантские switch-и. Внешний switch — по текущему состоянию автомата, внутренние — по событиям, на которые находясь в данном состоянии автомат может реагировать. Внутри — проверка условий стражников, выполнение действия по переходу и переход в следующее состояние. Состояние моделируется enumом, хранящимся как поле объекта. Для нашего примера получится что-то такое:

```

public void handleEvent(PanelEvent anEvent) {
    switch (currentState) {
        case PanelState.Open:
            switch (anEvent) {
                case PanelEvent.SafeClosed:
                    currentState = PanelState.Wait;
                }
                break;
        case PanelState.Wait:
            switch (anEvent) {
                case PanelEvent.CandleRemoved:
                    if (isDoorOpen) {
                        revealLock();
                        currentState = PanelState.Lock;
                    }
                }
                break;
        case PanelState.Lock:
            switch (anEvent) {
                case PanelEvent.KeyTurned:
                    if (isCandleIn) {
                        openSafe();
                        currentState = PanelState.Open;
                    } else {
                        releaseKillerRabbit();
                        currentState = PanelState.Final;
                    }
                }
                break;
    }
}

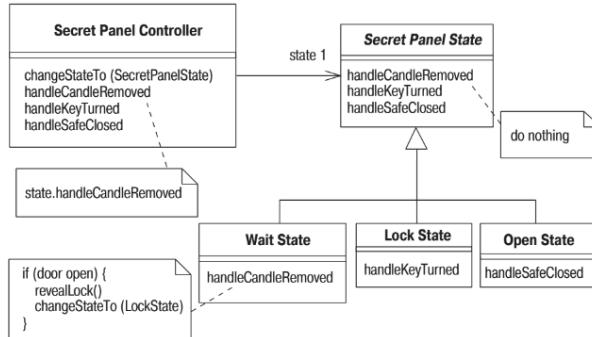
```

Это работает, но если этот код надо сопровождать, никто от него не будет в восторге. Для сколько-нибудь содержательных автоматов получается один метод на тысячи строк. Поэтому можно использовать таблицу состояний и универсальный интерпретатор, который просто ищет в таблице текущее состояние и событие, и выполняет то, что там написано. Вариантов таблиц состояний может быть много, но вот пример из книжки Фаулера:

Исходное состояние	Целевое состояние	Событие	Защита	Процедура
Wait	Lock	Candle removed (свеча удалена)	Door open (дверца открыта)	Reveal lock (показать замок)
Lock	Open	Key turned (ключ повернут)	Candle in (свеча на месте)	Open safe (открыть сейф)
Lock	Final	Key turned (ключ повернут)	Candle out (свеча удалена)	Release killer rab- bit (освободить убийцу-кролика)
Open	Wait	Safe closed (сейф закрыт)		

Таблица состояний обычно хранится как файл данных рядом с программой, либо вкомпилируется в исходный код.

Такой подход очень популярен в лексическом и синтаксическом анализе, но отлаживать или просто понимать такие программы очень тяжело. Поэтому есть ещё один, более объектно-ориентированный способ симулировать автомат: паттерн проектирования «Состояние»:

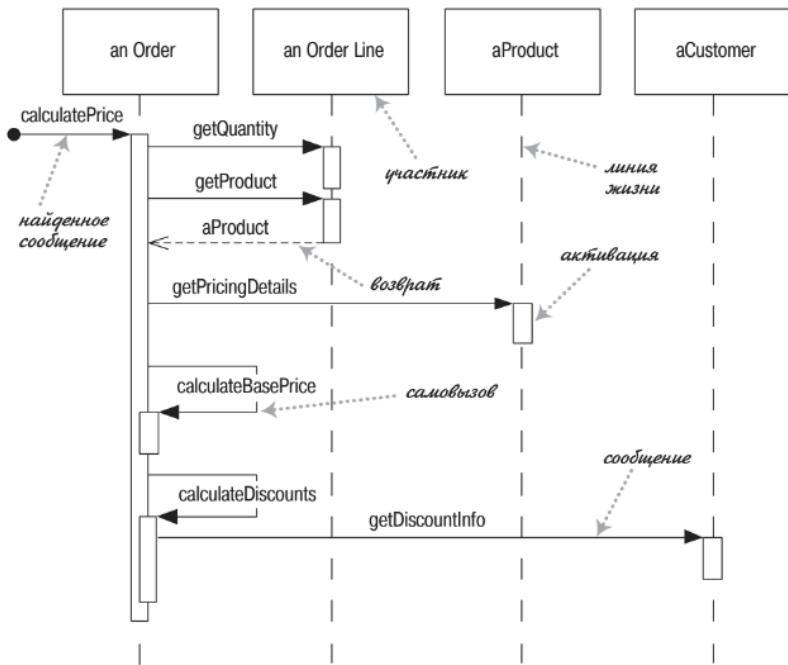


© М. Фаулер, UML. Основы

Автомат представляется в виде класса, который имеет в качестве поля ссылку на интерфейс «текущее состояние». Этот интерфейс имеет столько методов, сколько всего разных событий может обрабатывать автомат. Интерфейс реализуют конкретные классы, отвечающие за конкретные состояния, они определяют те методы интерфейса, на которые могут реагировать, там уже проверяют условия стражников и выполняют действие при переходе. Каждый такой метод возвращает объект-состояние, в которое должен перейти автомат дальше. То есть, по сути, это тот же switch, где самый большой switch (по состояниям) спрятан в таблицу виртуальных методов. Такой подход делает реализацию автомата более-менее читаемой, ограничивает ответственность каждого класса только одним состоянием и позволяет очень легко добавить новые состояния, поэтому очень популярен при «ручной» реализации автоматов.

### 3. Диаграммы последовательностей

Следующий тип диаграмм UML, использующийся при моделировании поведения — это диаграммы последовательностей (sequence diagrams). Они применяются для визуализации взаимодействия между объектами — передачи сообщений, возврата значений, времени жизни объекта. Выглядят диаграммы следующим образом:



© М. Фаулер, UML. Основы

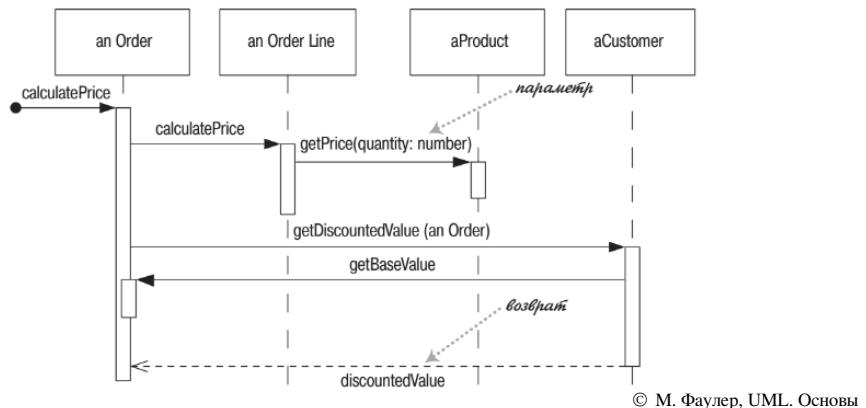
На диаграмме рисуются объекты (обратите внимание, не классы), из каждого объекта выходит *линия жизни* (пунктирная линия), на которой расположены *линии активации* (длинный белый прямоугольник). Линия жизни показывает, когда объект вообще существует в памяти, линия активации — когда объект занят какой-то работой (то есть работает либо его метод, либо метод, вызванный из его метода, или, более формально, какой-либо из методов объекта находится на стеке вызовов). Линии активации одновременно могут быть несколько — рекурсивные вызовы. Стрелки между линиями активации обозначают сообщения — как правило, это вызовы методов, и поэтому они ведут в начало соответствующих вызванному методу линий активации. Обратите внимание, что стрелка не может выходить из неактивного объекта и не может входить «в никуда».

Такая диаграмма позволяет разобраться в даже довольно сложных протоколах взаимодействия, поэтому применяется при многопоточном и асинхронном программировании, чтобы визуализировать общение между потоками/асинхронные вызовы. Также такие диаграммы очень популярны при описании телекоммуникационных протоколов (на самом деле, есть отдельный язык MSC (Message Sequence Charts), который применяется независимо от UML, но диаграммы последовательностей являются его почти копией).

Имеется и ряд менее очевидных применений:

- на этапе анализа предметной области для визуализации последовательности коммуникаций между участниками взаимодействия;
- на этапе проектирования для составления плана тестирования — кто в каком порядке должен дёргать и кто когда чем должен отвечать;
- на этапе отладки, для визуализации логов работающей системы.

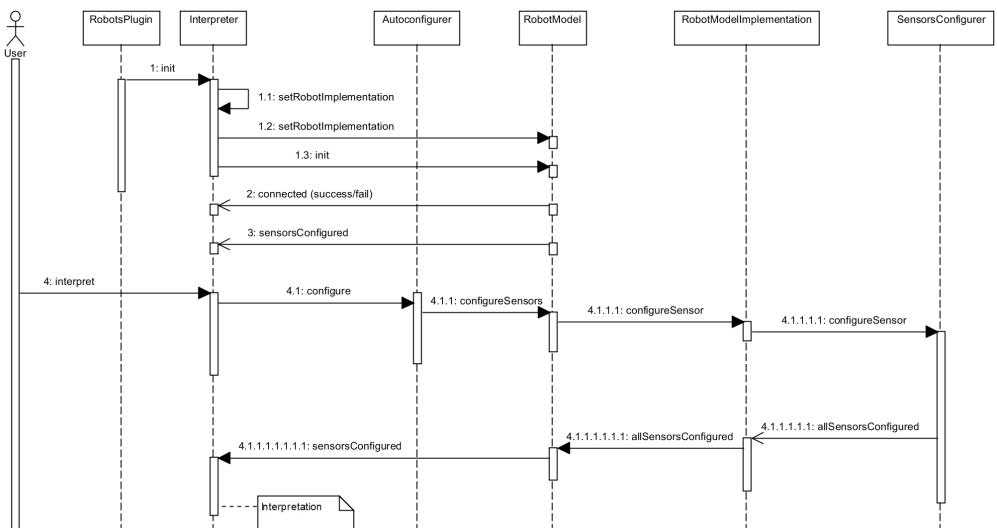
Ещё немного синтаксических подробностей:



На этом примере видна передача параметров в сообщении ( обратите внимание, что параметрами могут быть сами участники взаимодействия, например, «*an Order*» передаётся в «*aCustomer*», тот самый «*an Order*», из линии активации которого выходит сообщение). Здесь же показан возврат значения из вызова, «*discountedValue*».

### 3.1. Примеры

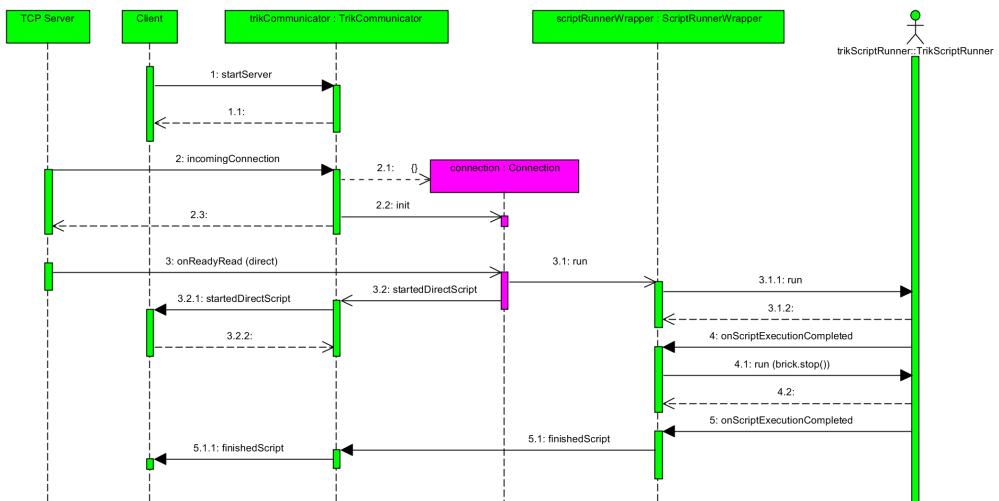
Пример диаграммы последовательностей из реального проекта:



Показывается сценарий конфигурирования робота при запуске интерпретации программы в среде программирования роботов. Сначала система инициализируется, выбирает реализацию подсистемы связи с роботом (исходя из того, с каким роботом сейчас работает пользователь), посыпает ему команду инициализации, после чего асинхронно полу-

чает два сигнала — подключение удалось/не удалось («connected») и сенсоры сконфигурированы на работе в конфигурации по умолчанию («sensorsConfigured»). Далее становится доступна кнопка «Interpret», пользователь нажимает её и запускает процесс конфигурации сенсоров — среда программирования смотрит на программу, понимает, какие сенсоры и как там используются, смотрит на настройки и рассчитывает итоговую конфигурацию сенсоров, которые и шлёт на робот. Дальше она дожидается асинхронного ответа о том, что все сенсоры готовы к работе, и начинает собственно интерпретацию программы. Как видим, протокол инициализации довольно сложен, но на диаграмме он вполне обозрим, и это очень помогает при отладке.

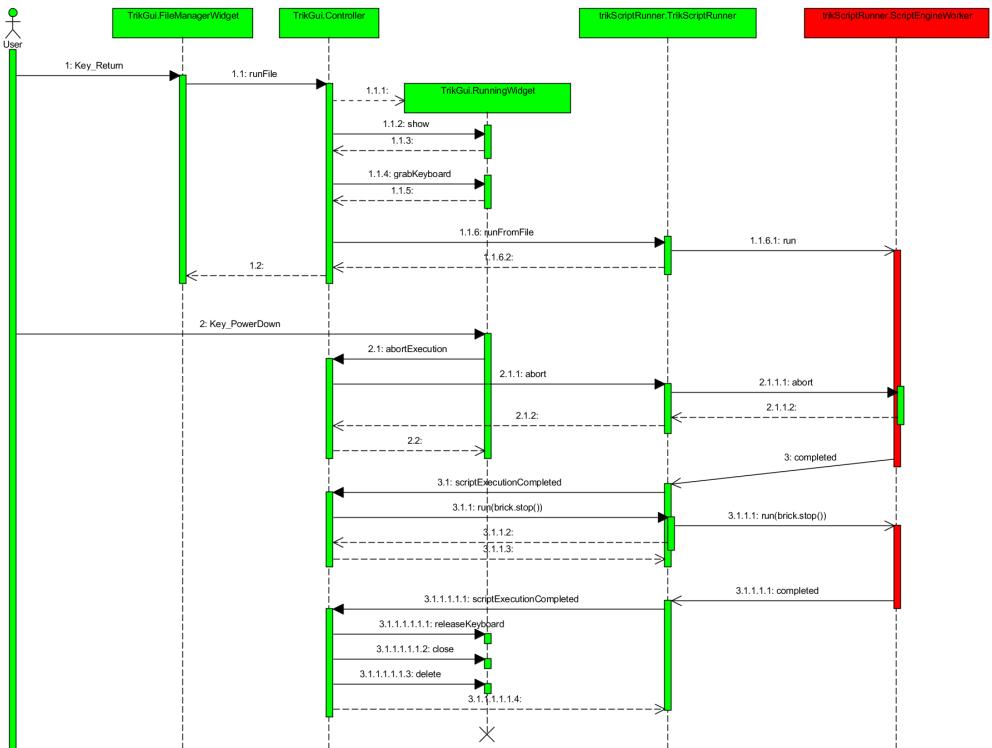
Ещё один пример:



На сей раз это сценарий выполнения скрипта на роботе. Скрипт получается по сети, для чего сначала запускают сервер, ждут входящего соединения, создают в отдельном потоке объект для его обработки, и, как только скрипт полностью загружен, запускают его на исполнение. Когда скрипт заканчивает работу, мы получаем сигнал «onScriptExecutionCompleted», после чего запускаем специальный скрипт остановки робота, который должен остановить все моторы и выключить все датчики. Уже после окончания этого скрипта мы посыпаем клиенту сигнал о том, что всё закончилось.

В этом примере цветом выделен объект, живущий в отдельном потоке, стандарт UML этого не предписывает (но и не запрещает), на практике очень удобно.

И ещё один пример:



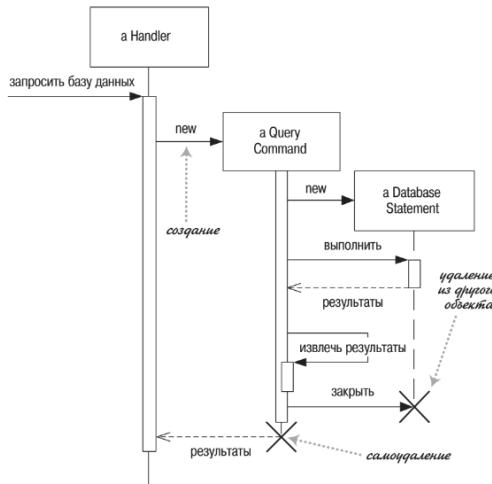
Это исполнение скрипта на роботе прямо с контроллера. Пользователь выбирает нужный скрипт в меню на экране робота, жмёт «return», после этого инициализируется виджет, отображающий работу скрипта, скрипт запускается в отдельном потоке. Пользователь хочет прервать работу до её естественного окончания, он жмёт на «powerDown», это посыпает команду «abort» в поток со скриптом, он выполняет согласованную остановку и шлёт обратно «completed». После этого запускается скрипт остановки робота, как в предыдущем примере. Как только он заканчивается, deinициализируем виджет исполнения скрипта и ждём следующие команды. Здесь тоже цветом выделен отдельный поток.

Все три примера взяты из проекта TrikStudio (<sup>1</sup>), где они показали себя как очень полезные для отладки и вообще понимания того, что происходит.

### 3.2. Подробности синтаксиса

На диаграмме также можно отобразить создание и удаление объекта:

<sup>1</sup> <https://github.com/trikset/trik-studio>



© М. Фаулер, UML. Основы

Создание означает, как правило, вызов конструктора, удаление более хитро — либо это вызов деструктора, либо то место, где объект становится больше не нужен и его может собрать сборщик мусора. Создание инициирует другой объект, удаление — либо сообщением от другого объекта, либо «самоубийство» в конце линии активации.

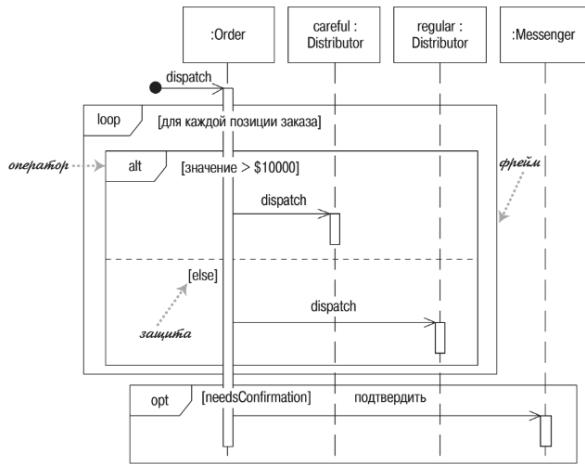
Все средства, описанные выше, позволяют задать только один конкретный сценарий взаимодействия, и обычно диаграммы последовательностей именно для этого и используются. Но если очень надо, можно попытаться отобразить целый алгоритм, с ветвленими и циклами, с помощью фреймов. Например, псевдокод

```

foreach (lineitem)
    if (product.value > $10K)
        careful.dispatch
    else
        regular.dispatch
    end if
end for
if (needsConfirmation)
    messenger.confirm

```

на диаграмме может выглядеть как



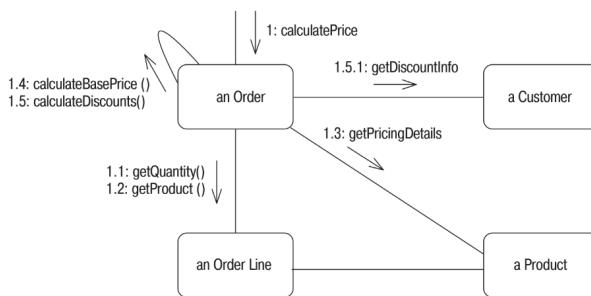
© М. Фаулер, UML. Основы

Фрейм ограничивает участок взаимодействия и позволяет выполнять его в цикле (фрейм «loop»), исполнять разные действия в зависимости от условия (фрейм «alt», который играет роль if или switch/case, поскольку может иметь несколько разделов со взаимоисключающими условиями), просто исполнять или нет в зависимости от условия (фрейм «opt»).

Вообще фреймы сильно ухудшают читабельность диаграммы, а для визуализации алгоритмов есть средства получше — диаграммы активностей, которые мы уже рассматривали, и диаграммы обзора взаимодействия, которые мы слегка рассмотрим дальше. Так что фреймы — очень ситуацияная штука.

## 4. Коммуникационные диаграммы

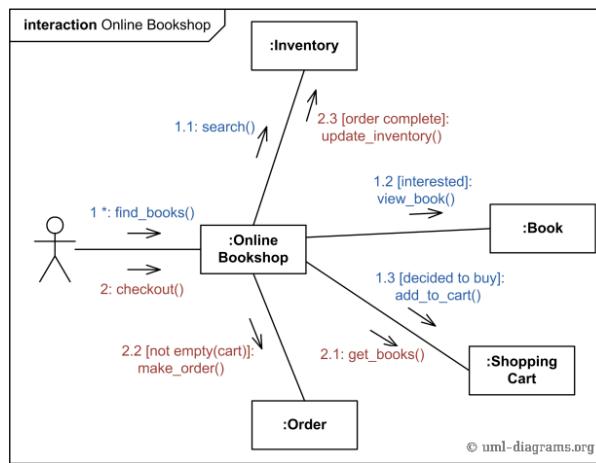
Коммуникационные диаграммы — это по сути те же диаграммы последовательностей, только «с высоты птичьего полёта». Они так же применяются для визуализации взаимодействия между объектами, так же подходят только для визуализации одного сценария взаимодействия и так же показывают последовательность обмена сообщениями:



© М. Фаулер, UML. Основы

Однако на диаграммах последовательностей поведение объектов рисуется снизу вверх, то есть, условно, по оси X откладываются объекты, по оси Y — время. На коммуникационных диаграммах объекты размещаются на двумерной плоскости, а порядок взаимодействия определяется числовыми индексами на стрелках. Это основное преимущество и основной недостаток таких диаграмм — они гораздо компактнее диаграмм последовательностей, но порядок действий во времени на них не очевиден.

Вот более содержательный пример, диаграмма коммуникаций для онлайн-магазина книг:

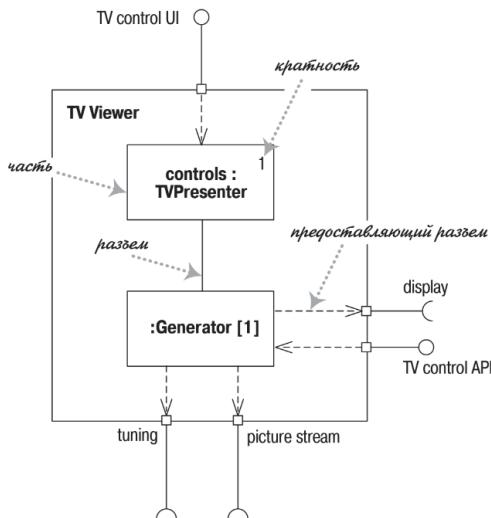


© <http://www.uml-diagrams.org/>

Сначала пользователь делает запрос `find_books()`, который обрабатывается объектом типа `Online Bookshop` (до двоеточия пишется имя объекта, если оно важно, после — имя типа, если оно важно). Вызов `find_books()` приводит к вызову `search()`, `view_book()` и `add_to_cart()` в именно такой последовательности. Обратите внимание на нумерацию запросов: 1.1 означает, что это первый вызов внутри вызова 1, 1.3 — третий вызов внутри вызова 1. Дальше пользователь выполняет запрос `checkout()`, который обрабатывается с помощью ещё двух вызовов.

## 5. Диаграммы составных структур

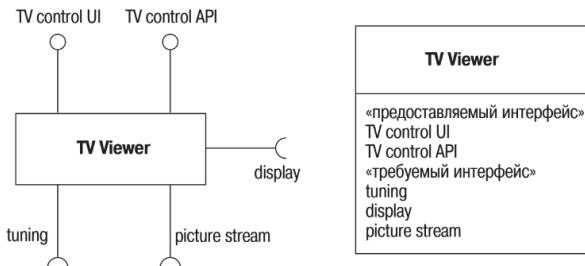
Диаграммы составных структур предназначены больше для проектирования аппаратного обеспечения, которое состоит из стандартизованных блоков, соединённых стандартными интерфейсами. По сути, диаграммы представляют собой продвинутые диаграммы компонентов — тут тоже рисуются крупные блоки, из которых состоит система, интерфейсы блоков, порты, связи. Однако же если диаграммы компонентов — это диаграммы, показывающие структуру времени компиляции, то на диаграммах составных структур внутри объемлющей компоненты не другие компоненты, а *роли*. Разница в том, что роль — это что-то вроде объекта, на диаграмме может быть несколько ролей одного типа, которые по-разному связаны друг с другом и делают разную работу. Так же, как объекты, роли имеют тип. В работающей системе роль может играть любой компонент правильного типа. Вот небольшой пример диаграммы:



© М. Фаулер, UML. Основы

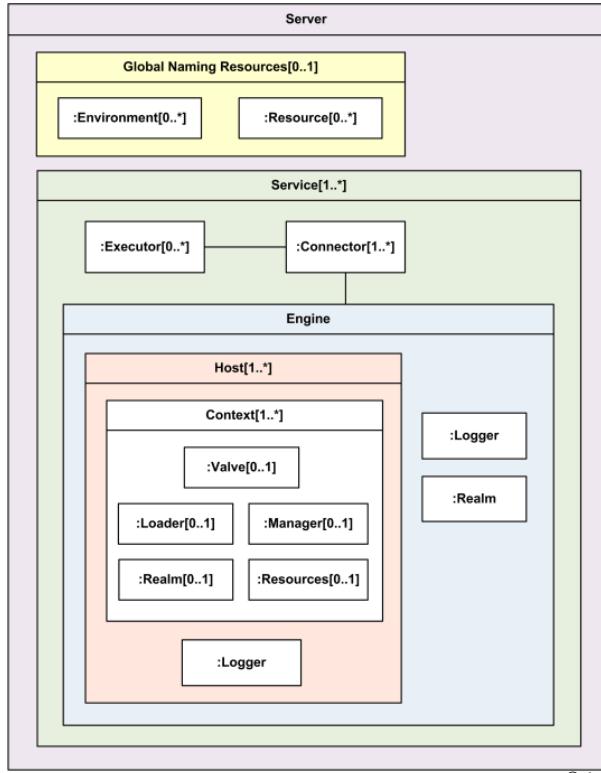
Тут controls и :Generator — это роли, в качестве controls может выступать любой компонент, реализующий интерфейс TVPresenter. [1] — это кратность компонента (и TVPresenter, и Generator в системе только один). Порты, интерфейсы и связи тут в целом такие же, как на диаграмме компонентов (только что связи называются “разъёмами”).

Интерфейсы компонентов можно описывать как в шарово-гнездовой нотации, так и внутри символа компонента:



© М. Фаулер, UML. Основы

Вот пример побольше, с <http://www.uml-diagrams.org/>:

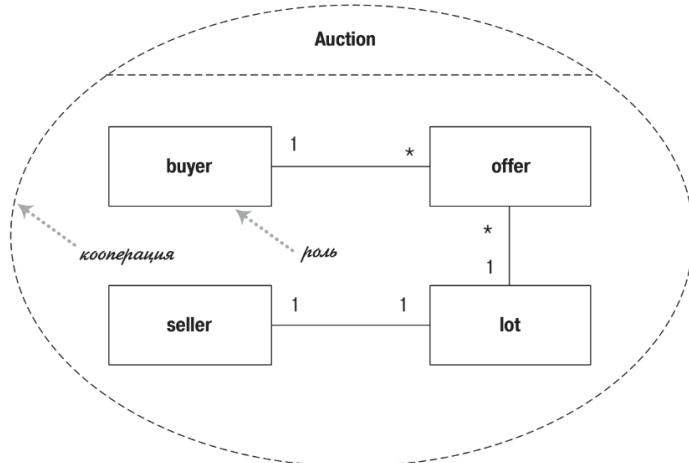


© <http://www.uml-diagrams.org/>

Выделение цветом в стандарте не прописано, но он его и не запрещает, поэтому этим часто пользуются (особенно на диаграмме компонентов и на диаграмме составных структур, где цвет удобен для визуализации иерархичности).

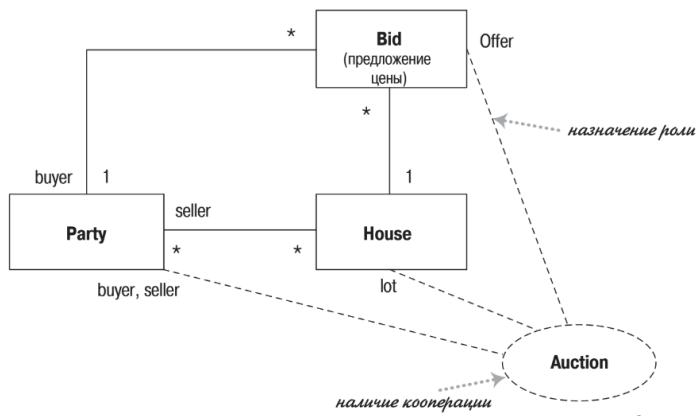
## 6. Диаграммы коопераций

Диаграммы коопераций — это что-то среднее между диаграммами объектов и диаграммами классов. Вместо классов и объектов на них рисуются *роли* — сущности, на место которых может быть подставлен настоящий объект. Диаграммы показывают взаимодействие ролей в рамках одного сценария использования, например:



© М. Фаулер, UML. Основы

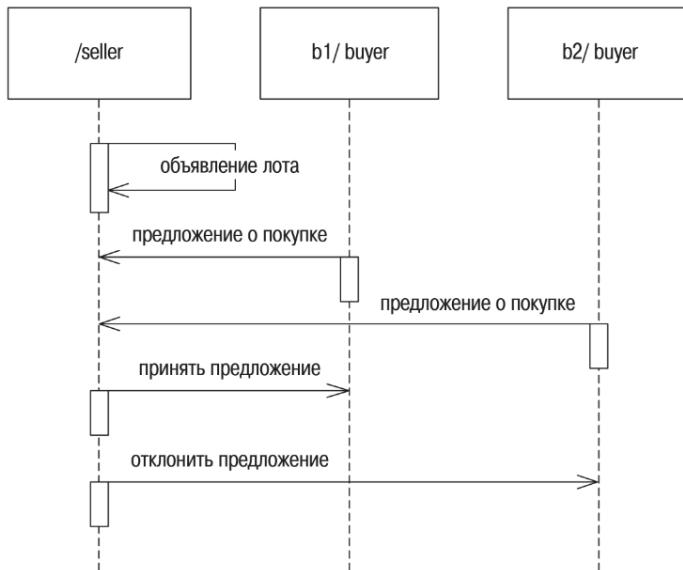
Либо, что то же самое,



© М. Фаулер, UML. Основы

Здесь показана кооперация в рамках сценария использования «Аукцион». Есть роли «покупатель», «продавец», которые могут взаимодействовать посредством лотов и предложений. Второй стиль рисования такой диаграммы больше похож на диаграмму классов, есть класс «сторона», исполняющий роли «покупатель» и «продавец» в рамках данного взаимодействия.

Нужны такие диаграммы, чтобы проиллюстрировать конкретный сценарий использования системы, либо чтобы сосредоточиться на анализе конкретного сценария. Эти диаграммы предоставляют простую структурную точку зрения на объекты, и могут дополняться диаграммами последовательностей для иллюстрации ещё и временных аспектов:

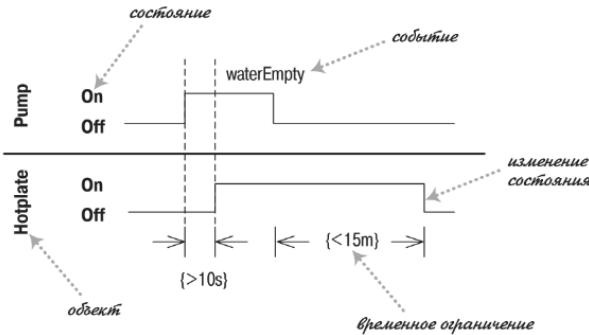


© М. Фаулер, UML. Основы

На диаграммах последовательностей в таких случаях пишут имя объекта «/» имя роли, которую этот объект играет в системе. Тут, например, видно, что покупателя в данном взаимодействии два.

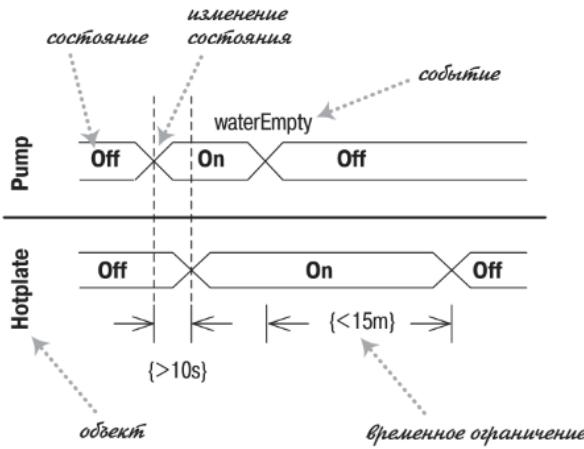
## 7. Временные диаграммы

Временные диаграммы создавались прежде всего для инженеров-электронщиков или для людей, проектирующих системы реального времени. Они нужны, чтобы визуализировать последовательности событий с чёткими временными ограничениями. Есть два варианта синтаксиса:



© М. Фаулер, UML. Основы

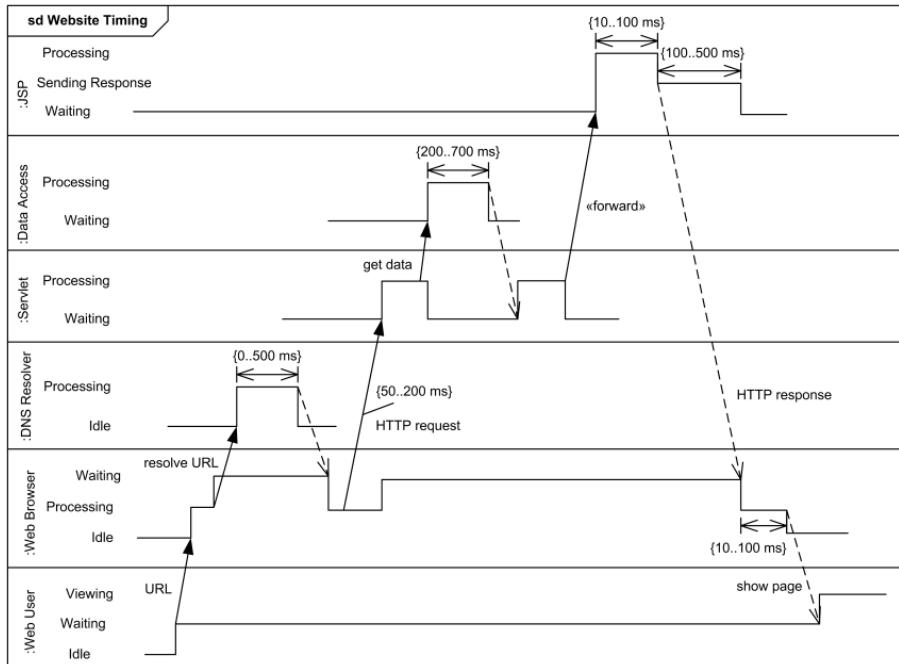
и



© M. Фаулер, UML. Основы

В обоих случаях рисуется временная шкала (время идёт слева направо), на ней вертикально располагаются объекты, которые могут находиться в некоторых состояниях. Первый вариант показывает переключение состояния как скачок линии, второй — как пересечение линий с именем состояния внутри. Второй вариант удобнее, если состояний много, первый вариант нагляднее. Помимо состояний указываются и временные ограничения, в фигурных скобках, как принято в UML для ограничений.

Вот более солидный пример, для запроса браузера:

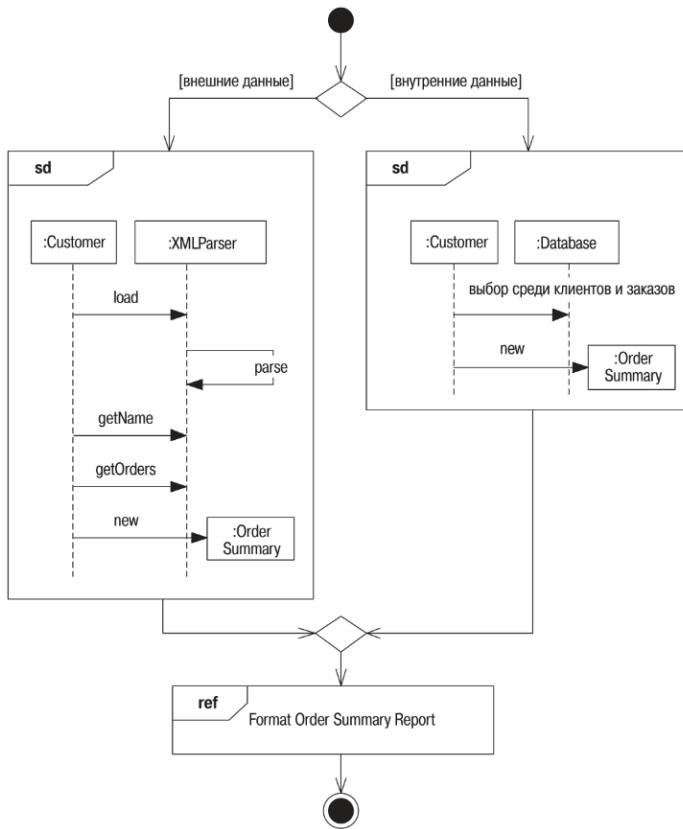


© <http://www.uml-diagrams.org/>

Видно, что диаграмма позволяет отразить последовательность событий столь же наглядно, как диаграмма последовательностей, но ещё и наглядно показывает временные ограничения, и гораздо нагляднее в плане последовательности изменений внутреннего состояния объектов.

## **8. Диаграммы обзора взаимодействия**

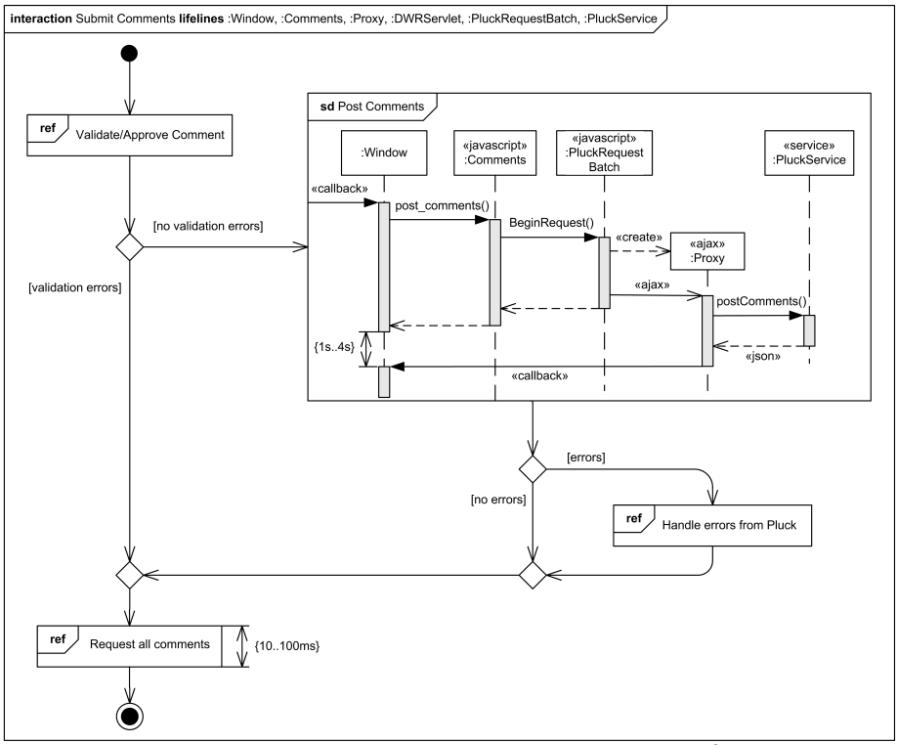
Диаграммы обзора взаимодействия — это на самом деле не более чем совмещённые на одной диаграмме элементы диаграммы последовательностей и диаграммы активностей. Выглядят они так:



© М. Фаулер, UML. Основы

Видно, что активность из диаграммы активностей может быть представлена в виде последовательности действий на диаграмме последовательностей. Применяются такие диаграммы, когда есть сложный алгоритм и нужна диаграмма последовательностей, которая бы его визуализировала, но имеется куча ветвлений и циклов. На диаграммах последовательностей есть фреймы, но для сколько-нибудь сложных алгоритмов они быстро сделают диаграмму нечитаемой — тут-то диаграммы обзора взаимодействия и пригодятся.

Вот более содержательный пример, валидация комментариев на сайте:



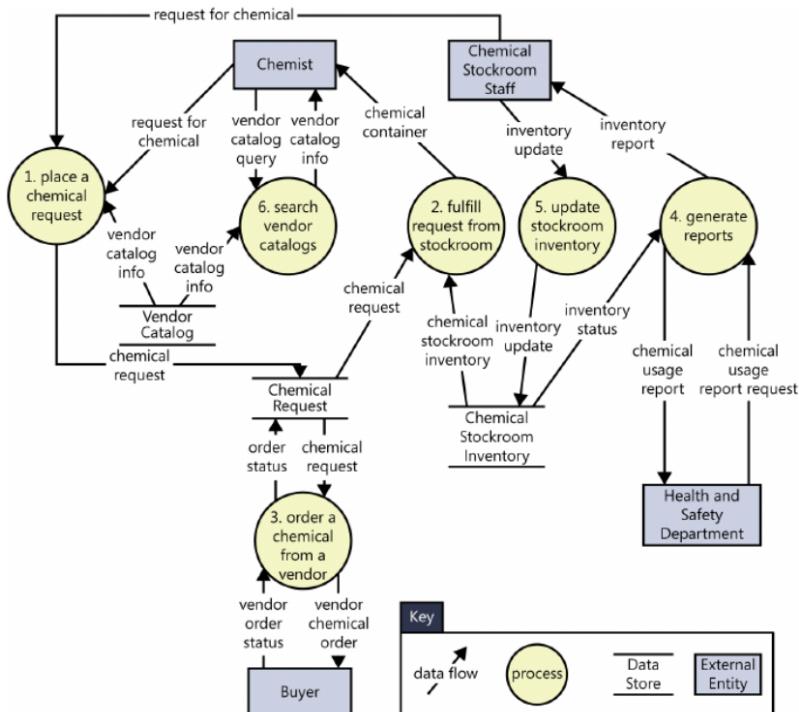
© <http://www.uml-diagrams.org/>

Как видно, на таких диаграммах тоже могут указываться временные ограничения.

## 9. Диаграммы потоков данных

На этом рассмотрение диаграмм UML закончилось. Но есть ещё полезные диаграммы, часто используемые сейчас при моделировании поведения программ, которые в UML не входят и даже не имеют там аналогов.

Первая такая диаграмма, пожалуй, самая популярная и самая древняя из них — это диаграмма потоков данных (Data Flow Diagram):



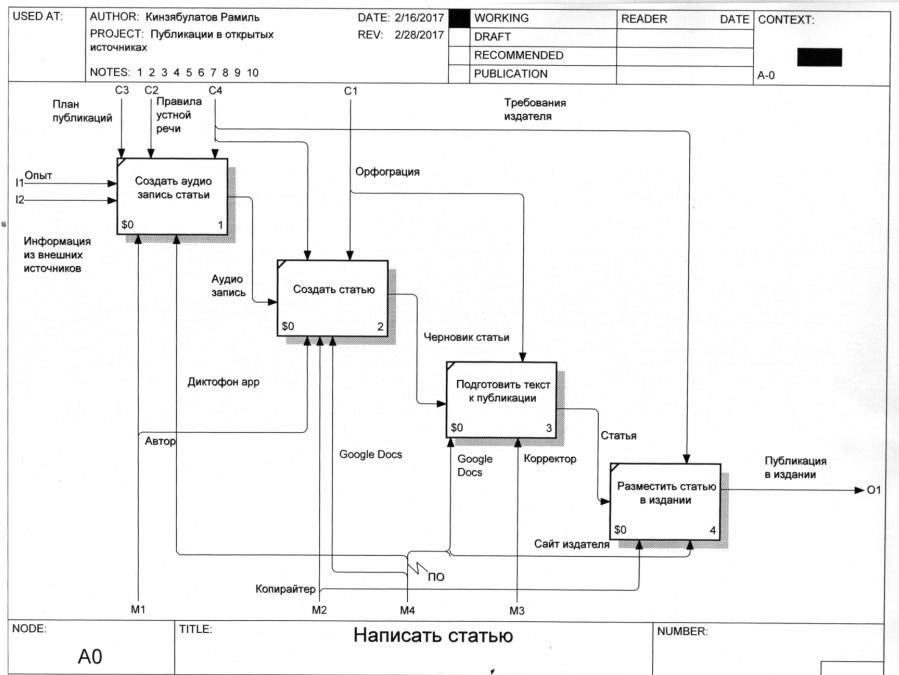
Синтаксис очень прост, есть один вид стрелок, который показывает поток данных, и три вида сущностей, между которыми, собственно, ходят данные:

- процесс — то, что может как-то преобразовывать данные внутри проектируемой системы, рисуется кружком;
- внешняя сущность — то, что поставляет или потребляет данные, рисуется прямоугольником;
- хранилище — то, где данные могут лежать, куда их можно поместить и забрать при необходимости, рисуется двумя горизонтальными линиями.

Такие диаграммы полезны как первый набросок архитектуры системы (например, когда есть бизнес-процесс, где данные уже как-то ходят) или как иллюстрация к уже созданной системе. Потоки данных между функциональными блоками обычно интуитивно понятны и вместе с тем довольно сложны, так что диаграммы потоков данных могут сообщить много информации человеку, который начинает знакомиться с системой.

## 10. Диаграммы IDEF0

Контекстные диаграммы IDEF0 в этом курсе уже упоминались, их дальнейшая детализация — диаграммы, описывающие декомпозицию системы и связи между её частями (которые, в свою очередь, тоже могут быть декомпозированы и раскрыты на отдельных диаграммах). Вот пример диаграммы IDEF0:



© <https://habrahabr.ru/post/322832/>

Диаграмма состоит из блоков (у каждого из них есть имя и однозначно идентифицирующий его номер) и стрелок. Стрелки, входящие в блок слева — материалы, которые блок перерабатывает в результаты (стрелки, выходящие справа). Сверху входят стрелки-управления — то, что регламентирует работу блока и как-то управляет им. Снизу входят стрелки-механизмы (или ресурсы) — это то, что блок непосредственно не перерабатывает, но необходимо блоку для работы. Например, блок может быть автозаводом, он перерабатывает запчасти в автомобили, как механизмы он использует станки и рабочих, как управление — государственные и отраслевые стандарты, заказы от дилеров.

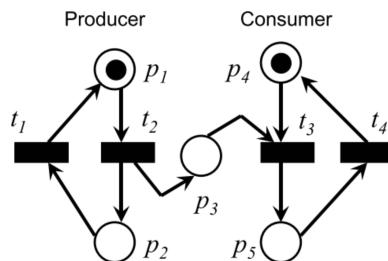
Над стрелками пишется, что из себя представляет эта стрелка, стрелки могут ветвиться (что может означать просто разделение ветки или детализацию — тогда над отдельившимися ветками пишутся уточнения надписи, которая была на исходной ветке) и сходиться в одну. Все ветки, входящие извне или исходящие с диаграммы, должны входить в блок или выходить из блока на диаграмме уровне выше, и так далее до контекстной диаграммы. При этом используется однозначная схема именования стрелок.

Такие диаграммы чаще используются в анализе бизнес-процессов; в разработке собственно ПО чаще применяются всё-таки диаграммы UML (диаграммы компонентов и диаграммы составных структур прекрасно заменяют IDEF0 в большинстве случаев). Тем не менее, аналитики IDEF0 любят и часто передают архитекторам как результаты своей работы, поэтому владеть IDEF0 архитекторам приходится.

## 11. Сети Петри

Сети Петри, в отличие от всего выше рассмотренного — это не просто визуальная нотация, а математический формализм, который просто обзавёлся визуальной нотацией потому, что это удобно. Сети Петри — ценный метод анализа параллельных программ, а поскольку подавляющее большинство современных систем так или иначе параллельны, сети Петри могут быть довольно полезны на практике (хотя в действительности ИТ-компаний сети Петри автор пока не встречал).

Итак, сеть Петри — это граф из мест, переходов и токенов, которые могут находиться в местах и прыгать через переходы. Простой пример сети Петри (для задачи производителя-потребителя):



Места рисуются кругами, переходы — закрашенными прямоугольниками, а токены — закрашенными кругами внутри мест. Токенов может быть много, как во всей диаграмме, так и в каждом месте. У диаграмм есть семантика — переход может *сработать*, если во всех его входных местах есть хоть один токен, и тогда он забирает по одному токену из каждого входного места и помещает по одному токену в каждое выходное место.

Так что в нашем примере готов к срабатыванию только переход  $t_2$  (для перехода  $t_3$  один токен есть, но ему два надо, в месте  $p_3$  токена нету). Когда переход  $t_2$  сработает, из места  $p_1$  токен исчезнет, зато появится по токену в местах  $p_2$  и  $p_3$ . И тогда уже  $t_3$  сможет сработать, но одновременно с ним может сработать и  $t_1$ . Считается, что из готовых переходов срабатывающий выбирается случайно, другая точка зрения на это — что все переходы срабатывают и не срабатывают одновременно, порождая из одного состояния множество состояний (как в недетерминированных автоматах). Теперь должно быть понятно, почему сеть из примера иллюстрирует задачу производителя-потребителя:  $p_3$  — это буфер с данными, Producer бегает в цикле, производит новые данные и кладёт их в буфер, Consumer, если буфер не пуст, бегает в цикле и забирает из буфера данные, параллельно с Producer. Токены в местах  $p_1$  и  $p_2$  симулируют текущую исполняемую инструкцию производителя, в местах  $p_4$  и  $p_5$  — текущую исполняемую инструкцию потребителя, токены в месте  $p_3$  — данные в буфере (их может накопиться сколь угодно много, если производитель работает быстрее).

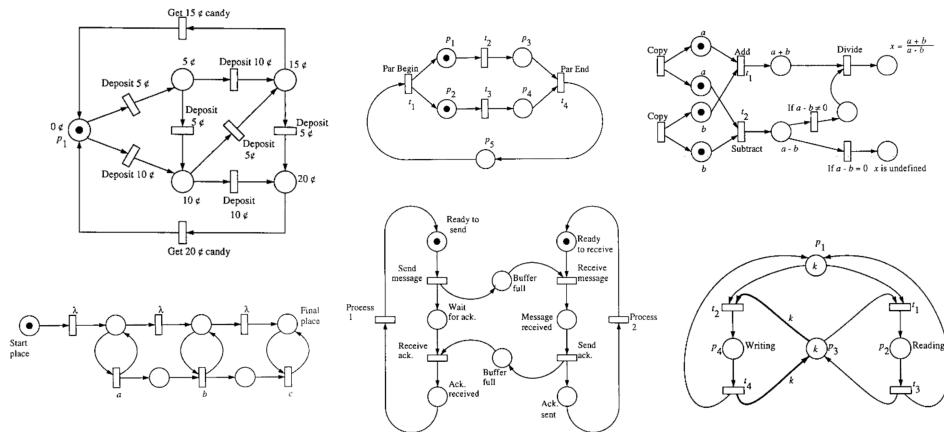
Однако диаграмма в случае с сетями Петри — это только вершина айсберга. Формально, сеть Петри — это:

- тройка  $(P, T, \phi)$ , где

- $P$  — множество мест;

- $T$  — множество переходов;
- $\phi$  — функция потока:  $\phi : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$  — определяет связи между местами и переходами ( $\mathbb{N}$  — множество натуральных чисел, число показывает, сколько токенов переход забирает, и сколько кладёт в место);
- маркировка:  $\mu : P \rightarrow \mathbb{N}$  — каждому месту сопоставляет количество токенов, которые в нём сейчас находятся;
- срабатывание (firing):  $\mu \xrightarrow{t} \mu' : \mu'(p) = \mu(p) - \phi(p, t) + \phi(t, p), \forall p \in P$  — собственно, переклазывает токены при срабатывании перехода.

Вот несколько примеров сетей Петри из хорошей, хоть и старой, обзорной статьи Murata Tadao «Petri nets: Properties, analysis and applications»:



© Murata Tadao. Petri nets: Properties, analysis and applications

Слева вверху, кстати, уже знакомый нам пример торгового автомата, который показывает, что вообще конечные автоматы — это частный случай сетей Петри (точнее, сеть, у которой каждый переход имеет ровно одно входное и ровно одно выходное место). Второй пример сверху показывает типичный параллельный кусок кода, который сначала разветвляется на два потока, потом потоки соединяются (например, функцией `join()`). Третий пример сверху показывает моделирование арифметической операции, а заодно и то, что у перехода может вообще не быть входных мест (тогда он просто бесконечно производит токены из ниоткуда). Кстати, может не быть и выходных — тогда переход «пожирает» все токены.

## 11.1. Свойства сетей Петри

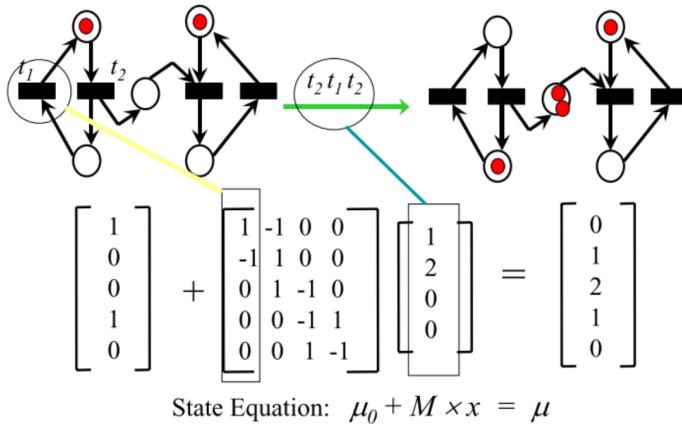
Зачем это всё вообще нужно: дело в том, что сети Петри с так определённой семантикой не тьюринг-полны, что, во-первых, означает, что не любая программа может быть точно смоделирована сетью Петри, а во-вторых, и в главных, что некоторые полезные свойства программ оказываются алгоритмически разрешимыми. Вот свойства, которые можно за конечное время доказать или опровергнуть для любой сети Петри:

- Поведенческие свойства (доказываемые для сети Петри с конкретной маркировкой, то есть распределением токенов по местам):
  - достижимость — что из заданной маркировки сети можно достичь другой заданной маркировки. Это означает, что можно доказать, что программа в принципе завершится (что невозможно в общем случае) или попадёт в заданное состояние.
  - Ограниченнность — что количество токенов в любом месте сети никогда не будет больше некоторого наперёд заданного числа, и более сильное свойство, *безопасность* — что количество токенов в любом месте сети никогда не будет больше 1, какие бы переходы в какой бы последовательности ни срабатывали (в программе никода не будет переполнения буфера).
  - Живость — целый набор свойств, гарантирующий разные степени утверждения «в программе нет дедлоков и она никогда не зависнет». *L0-живость* означает, что сеть «мертва», то есть ни один её переход никогда не сработает; *L1-живость* означает, что каждый переход сети может потенциально сработать, то есть существует такая последовательность срабатываний, когда переход срабатывает (в программе нет мёртвого кода); *L4-живость* означает, что любой переход может всегда сработать, то есть *L1-жив* в любой маркировке, достижимой из заданной (программа не может оказаться в состоянии, в которое не сможет вернуться, все состояния всегда достижимы).
  - «Реверсабельность» — из любого состояния, достижимого из исходного, мы всегда можем попасть в исходное; и более слабое свойство — «домашнее состояние» — когда из исходного достижимо такое состояние, в которое можно попасть из любого состояния, которое из него достижимо. Реверсабельность в программах означает, что мы всегда можем вернуться к исходному состоянию, что бы мы ни делали, а «домашнее состояние» означает, что сначала программа может как-то инициализироваться, выполнять дополнительную установку и т.д., но как только она готова к работе, она, например, показывает стартовый экран, на который мы всегда можем вернуться.
  - и т.д., есть ещё менее важные на практике, но интересные свойства, которые можно доказывать, см. статью M. Tadao.
- Структурные свойства (доказываемые для сети Петри с **любой** маркировкой):
  - структурная живость — существует хотя бы одна маркировка, при которой сеть *L1-жива* (нет мёртвого кода);
  - полная контролируемость — любая маркировка достижима из любой другой маркировки (что бы мы ни делали с программой, мы всегда можем попасть в любое её состояние за конечное число шагов);
  - структурная ограниченность — что при любой конечной начальной маркировке количество токенов в каждом месте не превосходит некоторого наперёд заданного для данной маркировки числа (в программе в принципе невозможны переполнения буфера);

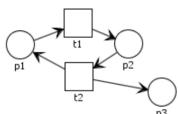
- консервативность — количество токенов в сети константно (программа корректно обрабатывает физические ресурсы, которые нельзя просто раскопировать);
- и т.д. и т.п.

## 11.2. Способы анализа

Доказывать вышеприведённые свойства для данной сети можно разными способами. Самый алгоритмически эффективный, но вместе с тем самый слабый способ анализа — это алгебраический. Посмотрим на состояние сети Петри как на вектор длины  $|P|$ , где на  $i$ -й позиции будет стоять число токенов в месте  $i$ . И определим матрицу переходов  $M$ , где  $M_{i,j}$  будет равно количеству токенов, которые  $i$ -й переход кладёт в  $j$ -е место ( $M_{i,j}$  больше нуля для выходных мест перехода и меньше нуля для входных). Тогда просимулировать состояние сети после нескольких переходов мы можем, прибавив вектор начального состояния к матрице переходов, умноженной на вектор срабатываний (вектор размерностью  $|T|$ , где на  $i$ -й позиции стоит число срабатываний перехода  $i$ ). Например:



Матрицу переходов легко построить по данной сети (и она зависит только от топологии сети), вектор начального состояния легко строится по начальной маркировке (существенно, он и есть начальная маркировка). Желаемое конечное состояние обычно известно, так что мы можем записать уравнение, где неизвестным будет вектор переходов. Если такое уравнение не решается, то конечное состояние точно недостижимо из начального (а решать системы линейных уравнений можно за полиномиальное время). Однако если такое уравнение решается, это, внезапно, ещё ничего не значит. Простой пример:

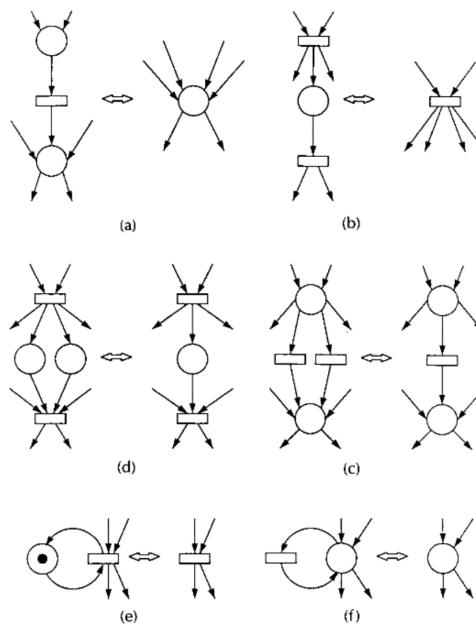


$$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -1 & 1 \\ 1 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Тут вектор  $(1, 1)$ , очевидно, является решением уравнения состояния, но сеть, изначально не имеющая маркеров, очевидно, маркер произвести никак не может. Почему так

— потому что уравнение состояния никак не учитывает последовательности срабатываний, а следовательно, и живости переходов в момент, когда по мнению уравнения состояния переход должен сработать. Если бы в примере выше переходы реально могли сработать, сеть бы оказалась в нужном состоянии без проблем, но они не могут.

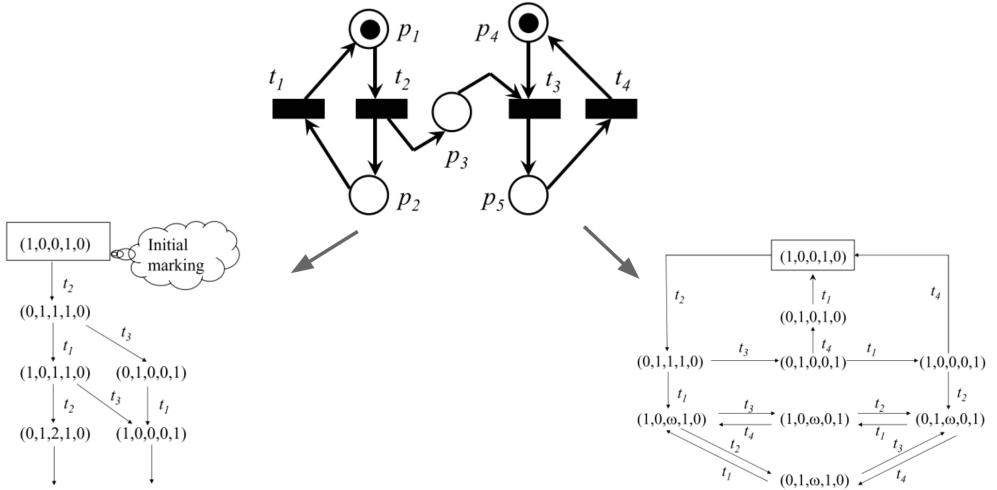
Поэтому алгебраический способ подходит только, чтобы быстро проверить, может ли выполняться доказываемое свойство вообще. Если да, то может применяться структурный анализ — это когда мы выделяем в сети некоторые структурные элементы (например, *цифоны*, то есть куски сети, которые производят токены, и *ловушки*, которые токены поглощают), и смотрим, как они соотносятся друг с другом. Либо анализ редукцией — когда мы упрощаем сеть, склеивая состояния и переходы, которые можно склеить, по следующим правилам:



© Murata Tadao. Petri nets: Properties, analysis and applications

Этот способ сам по себе редко что-то даёт, но его имеет смысл применять, чтобы упростить сеть и тем самым уменьшить размер задачи для самого мощного и самого трудоёмкого способа анализа сети: метода анализа пространства состояний.

Метод заключается в построении дерева (или графа) всех возможных состояний, которые могут быть получены переходами из данного:



Из каждого состояния (то бишь маркировки сети) ведут рёбра, соответствующие живым переходам в новое состояние (то бишь маркировку, полученную переходом). Дерево обладает тем неприятным (но ожидаемым) свойством, что оно бесконечно для подавляющего большинства сетей, поэтому чаще рассматривают граф переходов. В графе неограниченные состояния (то есть состояния, в которых в одном из мест может быть сколько угодно токенов) записываются с использованием символа  $\omega$ . Например, из состояния  $(0, 1, 1, 1, 0)$  можно попасть в  $(1, 0, 1, 1, 0)$ , из него в  $(0, 1, 2, 1, 0)$ . И тут мы замечаем, что  $(0, 1, 2, 1, 0)$  покрывает состояние  $(0, 1, 1, 1, 0)$ , то есть в нём количество токенов во всех местах сети больше либо равно, чем в состоянии  $(0, 1, 1, 1, 0)$ . Это означает, что во всех местах, в которых токенов строго больше, может быть бесконечно много токенов (в нашем примере это соответствует ситуации, когда производитель производит данные быстрее, чем потребитель успевает их обрабатывать), так что все такие состояния можно склеить в одно, считая количество токенов в «плохих» местах равным  $\omega$  (которое означает «сколь угодно много»). Граф состояний всегда конечен.

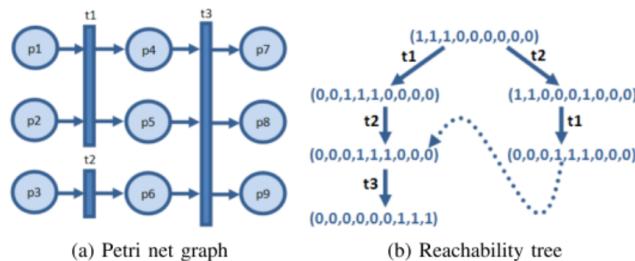
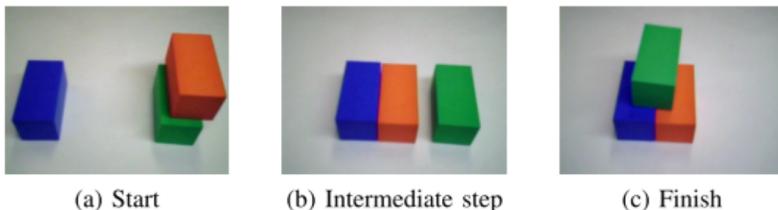
Анализ пространства состояний позволяет точно доказать или опровергнуть очень многие свойства сети, но имеет один важный недостаток — экспоненциальный рост числа маркировок при росте числа мест и переходов сети (даже в случае использования графа, который не так-то тривиально построить). Но по сравнению с машинами Тьюринга, для которых любое нетривиальное свойство скорее всего вообще алгоритмически неразрешимо, это считается успехом, и в сообществе, занимающемся сетями Петри, искренне радуются новым алгоритмам с экспоненциальной трудоёмкостью.

## 11.3. Примеры использования

Закончим мы парой примеров использования сетей Петри в программировании, взятых из научных статей (вообще, в научном сообществе сети Петри всё ещё очень популярны, несмотря на то, что, казалось бы, всё, что про них можно было сказать, было сказано ещё в начале 90-х).

Первый пример из статьи G. Chang, D. Kulić «Robot Task Learning from Demonstration Using Petri Nets» 2013-го года. Там сети Петри использовались как способ хранения и

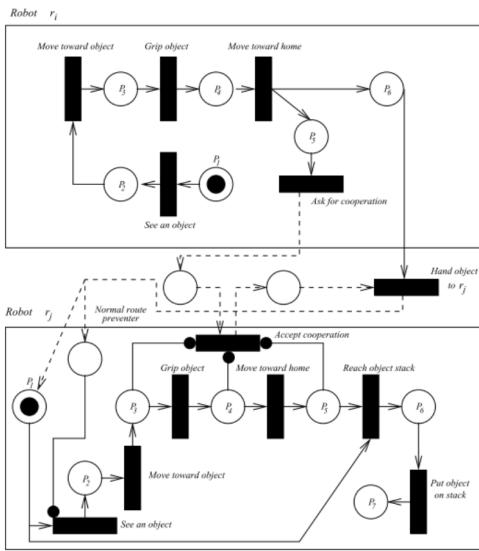
представления знаний. Роботу с камерой показывают, как выполнять то или иное действие (например, переложить цветные параллелепипеды в правильном порядке), он строит сеть Петри, соединяющую состояния наблюдаемых объектов и действия (переходы). Потом воспроизводит последовательность действий от начального состояния к заданному. При этом, в чём тут преимущество сетей Петри, робот может сам найти решение похожей задачи (или подзадачи) путём поиска по дереву состояний:



© G. Chang, D. Kulić. Robot Task Learning from Demonstration Using Petri Nets

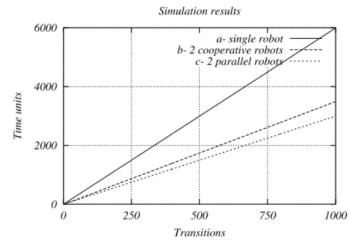
Состояний обычно не очень много, так что перебор пространства состояний не занимает много времени. И позволяет решить проблему, сформулированную в анекдоте «Как программисту вскипятить чайник? Налить воды, поставить на огонь, подождать пока вскипит. А если в чайнике уже есть вода? Вылить воду и свести задачу к предыдущей.».

Второй пример — из статьи 2007 года Y.T. Kotb et al. «Petri Net-Based Cooperation In Multi-Agent Systems». Это тоже была статья про роботов, но на сей раз роботов много и все они обладают разными возможностями — например, у кого-то есть манипулятор, кто-то может перевозить грузы. Задача — обеспечить совместное достижение группой роботов некоторой цели, с автоматическим распределением задач между роботами. Решение — представим деятельность каждого робота в виде сети Петри, где переходам будут соответствовать действия, а местам — состояния роботов. Заведём промежуточные места и переходы, отвечающие точкам синхронизации роботов, и будем искать на пространстве состояний пути, которые быстрее всего приведут к целевому состоянию:



	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$	$T_7$	$T_8$	$T_9$	$T_{10}$	$T_{10'}$
$a_1$	1	0	0	0		1	0	0	0	0	0
$a_2$	0	1	0	0		0	1	0	0	0	0
$a_3$	0	0	1	0		0	0	1	0	0	0
$a_4$	0	0	0	1		0	0	0	1	0	0
$a_5$	0	0	0	0		0	0	0	0	1	0
$a_6$	0	0	0	0		0	0	0	0	0	1

Transitions that belong to Petrinet of robot  $r_i$       Transitions that belong to Petrinet of robot  $r_j$



© Y.T. Kotb et al. Petri Net-Based Cooperation In Multi-Agent Systems

## 12. Литература

На этом заканчивается обзор языков визуального моделирования. Для закрепления знаний очень рекомендую книжку М. Фаулер, UML. Основы. Краткое руководство по стандартному языку объектного моделирования. СПб., Символ-Плюс, 2011. 192 С. Книжка короткая, состоит в основном из картинок, но содержит, в целом, все знания, которые нужны, чтобы использовать на практике UML-диаграммы.



# Лекция 9: Антипаттерны

Юрий Литвинов

yurii.litvinov@gmail.com

## 1. Введение

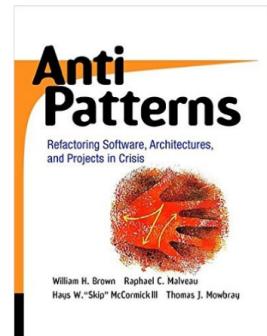
В этой лекции будет рассказано про то, как делать не надо. В литературе есть термин “Антипаттерны”, имеющий тот же смысл, что и паттерны, но наоборот: если паттерны — это часто встречающиеся решения, приводящие к известным преимуществам, то антипаттерны — это часто встречающиеся решения, приводящие к известным проблемам. Оказывается, что знание антипаттернов для архитектора даже полезнее, чем знание паттернов, поскольку позволяет избежать очень частых и очень дорогостоящих ошибок.

Точно так же, как и паттерны, антипаттерны документируют, чтобы иметь общий словарь (например, не пускаться в пространные рассказы про то, что есть принцип единственности ответственности, что класс, который делает вообще всё — это плохо, и почему, а просто сказать, что это антипаттерн “God Object”). Кроме того, описание антипаттерна, помимо описания проблемы и пояснения, почему это плохо, должно по-хорошему содержать и рекомендации по тому, как это исправить.

Хочется заметить, что антипаттерны часто описывают решения, которые сами по себе неплохи и вполне применяются на практике без всяких проблем, типичный пример — это “Busy waiting”, успешно применяющийся в микроконтроллерах. Источником проблем часто является не само решение, а контекст его применения — например, “Busy waiting” может быть очень плохой идеей, если минутами грузит процессор на ноутбуке.

Антипаттерны, так же, как и паттерны, имеют некую классификацию, но не по назначению, а по “сфере применения” — антипаттерны реализации (в том числе, специфичные для конкретного языка или технологии, которые мы в этой лекции затрагивать не будем), архитектурные антипаттерны и организационные антипаттерны (которые мы тоже затрагивать практически не будем, потому что это больше про управление проектами, чем про архитектуру). Мы начнём с антипаттернов реализации и продолжим архитектурными антипаттернами.

Дальнейшее изложение будет вестись по книжке AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis by William J. Brown, Raphael C. Malveau, SkipMcCormick, Thomas J. Mowbray, Wiley, 1998, 336pp. Книга столь же старая, что и книга про паттерны, и с момента её издания появилась куча специфичных для языков и технологий антипаттернов, но они нам как раз и не очень интересны. Антипаттерны, касающиеся разработки в целом, за это время нисколько не поменялись. Книга не входит в рекомендованную по этому курсу литературу, потому что там много воды и наиболее содержательные части всё равно будут в этой лекции рассказаны, так что читайте только если особо интересно.



## 2. Семь причин провала проектов

Авторы книжки про антипаттерны вделяют семь главных причин, по которым эти антипаттерны, собственно, вообще появляются в промышленном коде. Эти причины таковы.

- Спешка — пожалуй, самая главная причина появления плохого кода. Бывает тяжело думать над правильным и архитектурно красивым решением, если завтра релиз, а ещё не поправлен десяток критических багов. Проблема в том, что промышленная разработка почти всегда ведётся в спешке, и на это есть простые экономические причины — если у программистов есть время делать хорошо и правильно, они могли бы потратить это время на то, чтобы сделать больше фич или сдать больше проектов, поэтому никто, кроме самих программистов, не заинтересован в качестве кода. Заказчику не нужна архитектура, ему нужен работающий продукт. И если даже иногда оно будет вести себя странно, это тоже может быть некритично — пользовательerezапустит приложение, и всё. Из-за этого растёт технический долг — вы думаете, что завтра релиз и послезавтра вы перепишете все костыли, что сейчас навставляли, но послезавтра новый релиз через две недели, на вас уже написали задач, с которыми вы чувствуете, что справитесь только недели через три, и цикл повторяется. И получение позитивной обратной связи в виде довольного менеджмента и заказчика только ухудшает ситуацию.
- Апатия — неприменение известных хороших решений, потому что, казалось бы, незачем. Зачем заботиться о переиспользовании, если код всё равно пишется на выброс? Зачем заботиться о производительности, если всё равно потом надо будет оптимизировать код? Зачем думать о расширяемости, если проект всё равно закончится через два месяца? Однако оказывается, что оценки автора кода, касающиеся сроков его жизни, могут отличаться от истины в сотни раз, хороший пример тому — проблема 2000 (Y2K Problem), когда авторы приложений 80-х хранили год двумя цифрами, будучи уверенными, что уж до 2000-го года их приложения сотню раз заменят на новые.

- Недалёкость — незнание хороших решений. Незнание само по себе не является проблемой, скорее, это обычная ситуация в программной инженерии, одних только библиотек для frontend-разработки появляется по нескольку каждый месяц, и знать всё невозможно. Проблема — нежелание погуглить. Ну и незнание, что гуглить — если вы не имеете представления о том, что вообще бывает, то и интернет вам не поможет (для чего, кстати, и нужно современному специалисту университетское образование — знать, что вообще искать в интернете, и уметь выбрать нужное).
- Лень — движитель прогресса, если её правильно использовать, но если использовать неправильно, то причина появления дурацкого кода, минимальными усилиями решающего поставленную задачу в минимальной её интерпретации.
- Жадность — наоборот, желание сделать больше работы, чем надо (например, из желания продвинуться по служебной лестнице или просто неумения делегировать). Хороший пример — архитектурная жадность, когда архитектор специфицирует структуру системы с точностью до каждого параметра каждого метода каждого класса. Архитектура выглядит очень детальной, архитектор выглядит очень крутым, но в ходе реализации выясняется, что это всё не имеет отношения к реальной жизни. А если архитектор будет настаивать, то результат получится гораздо хуже, чем если бы излишние подробности доверили программистам.
- Неведение — нежелание понимать причины и смысл чужих решений. Например, этим страдают звёздные программисты, начинающие рефакторить под свой любимый стайлгайд весь существующий код проекта, или архитекторы, которые считают, что они архитекторы и поэтому правы.
- Гордость — нежелание переиспользовать готовые решения. Знаменитый синдром “Not invented here” можно отнести именно к гордости. Иногда это неплохо, во избежание привязки проекта к конкретному поставщику конкретной библиотеки или просто из соображений владения кодом и возможности модифицировать его под свои цели. Иногда это очень плохо, поскольку в разы увеличивает трудозатраты при реализации проекта, да и результат получается существенно хуже, чем мог бы быть.

### 3. Антипаттерны реализации

Рассмотрение собственно антипаттернов мы начнём с антипаттернов реализации — частых структур, встречающихся в коде проекта, которые имеют свои корни в неудачной низкоуровневой архитектуре.

#### 3.1. Circular dependency

Circular dependency, или “круговая зависимость” — это когда два или больше компонентов зависят друг от друга циклически, например, первый зависит от второго, второй от третьего, а третий от первого. Появляется чаще всего от незнания принципа Dependency Inversion или от неудачных попыток добавить callback-и, чтобы организовать реально двухстороннее общение компонент. Обратите внимание, что компоненты, обменивающиеся данными в обе стороны — это не Circular dependency, это нормально. Circular dependency

— это круговая зависимость во время компиляции или круговая зависимость в графе вызовов. Особо опасна круговая зависимость заголовочных файлов в C++ — положим у нас есть файл a.h такого содержания:

```
#pragma once
```

```
#include "b.h"
```

```
class A {  
public:  
    B b;  
}
```

И файл b.h:

```
#pragma once
```

```
#include "a.h"
```

```
class B {  
public:  
    A a;  
}
```

При использовании этих файлов мы неизбежно получим ошибку “неизвестный идентификатор” либо на поле типа A, либо на поле типа B. И это может повергнуть в шок начинающего C++-программиста, потому что объявление — вот оно, include написан, IDE всё подсвечивает и показывает правильно, а компилятор недоволен. Почему — стражи включения (та самая pragma once) заставляют препроцессор подключать каждый файл только один раз, при подключении a.h подключится b.h, который захочет подключить a.h, но не сможет, потому что тот файл препроцессор уже видел, ну и в классе B объявление класса A будет не видно (файл подключен, но “ниже” по коду).

Как бороться: прежде всего применением принципа Dependency Inversion, пятого принципа SOLID. Если есть круговая зависимость, сделайте так, чтобы обе абстракции, зависящие друг от друга, зависели от интерфейсов и реализовывали соответствующие интерфейсы. Также тогда можно будет применить Dependency Injection для автоматического внедрения зависимостей и конфигурирования системы, что не обязательно, но приятное дополнение к более аккуратной архитектуре.

При аккуратной реализации велика вероятность, что круговая зависимость исчезнет сама собой, но вполне возможно, что абстракции объективно зависят друг от друга. Тогда, возможно, потребуется более аккуратное проектирование. Поскольку чаще всего круговые зависимости связаны с необходимостью вызова колбэков, может помочь паттерн «Наблюдатель» — чтобы один компонент подписывался на события другого, а не дёргался другим напрямую. Ещё может помочь более аккуратно разбить функциональность на слои, чтобы вызовы могли выполняться только от верхних слоёв к нижним (и использовать «Наблюдатель» в обратную сторону), но это сложно сделать пост-фактум. Зато если вы изначально проектируете систему в слоистом стиле, вероятность появления круговых зависимостей будет существенно меньше.

### 3.2. Sequential coupling

Sequential coupling, или «последовательная связность» — необходимость вызывать методы в строго определённом порядке, ошибившись в котором можно всё сломать, или даже просто необходимость обязательно вызвать второй метод, если вызван первый. Антипаттерн это потому, что компилятор не в состоянии это проверить, а всё, что не проверяется компилятором, рано или поздно сделают не так.

Самый распространённый пример — метод `init()`, который обязательно надо вызывать после конструктора объекта, иначе тот будет в неконсистентном состоянии. Когда-нибудь его забудут вызвать практически наверняка. Однако гораздо более опасный случай — это последовательность инициализации крупного приложения, например, CASE-системы — сначала надо создать репозиторий, потом палитру, потом сцену, подписать палитру и сцену на события репозитория, загрузить сохранение, внезапно выяснить, что пока мы грузим сохранение, репозиторий шлёт массу событий изменения, которые ловятся сценой и она пытается отрисовать куски диаграммы, понять, что это какая-то фигня и отключить посылку репозиторием событий на время загрузки сохранения, потом забыть его включить и увидеть пустую сцену и т.д. События, колбэки и нотификации через «Наблюдатель» делают последовательную связность крайне болезненной, потому что в системе что-то происходит «само собой» и должно обязательно происходить в правильном порядке, а то какое-то событие пошлётся до того, как его будут готовы обработать.

Решения проблемы последовательной связности в порядке от самых тактических до самых стратегических:

- проектировать интерфейсы абстракций так, чтобы все операции были «атомарны», то есть делали всю работу за раз;
- использовать паттерн «шаблонный метод», где инкапсулировать эту самую необходимую последовательность, чтобы прикладные программисты о ней не думали;
- использовать паттерны «абстрактная фабрика» или «строитель», чтобы там спрятать сложность инициализации подсистемы в правильном порядке;
- использовать Dependency Injection, чтобы переложить порядок инициализации вообще на стороннюю библиотеку — но помните, что Dependency Injection имеет и свои недостатки, связанные со сложностью отладки и отсутствием контроля во время компиляции.

### 3.3. Call Super

Call Super, или «вызов предка» — это необходимость вызвать из переопределённого метода потомка переопределяемый метод предка. Например, так часто бывает в библиотеках пользовательского интерфейса — в переопределённом методе `paint()` надо обязательно вызвать `paint()` у родителя, иначе контрол отрисуется наполовину, то же с обработкой событий — забыли вызвать обработчик предка из переопределённого, ваше поведение будет работать, поведение по умолчанию нет. Плохо это, как и в предыдущем антипаттерне, потому, что компилятор не может проверить, вызвали вы метод предка или нет.

Для борьбы с этим антипаттерном особо эффективен «шаблонный метод» — тут как раз уже есть предок, где его можно разместить. Шаблонный метод **позволяет** переопределить поведение родителя, а не **требует** этого, к тому же предоставляет фиксированные точки расширения, позволяющие выполнить это определение и точно ничего не сломать. Это, конечно, уменьшает гибкость (и поэтому в GUI-библиотеках так обычно не делают), но увеличивает надёжность системы (и поэтому в GUI-библиотеках так всё-таки делают иногда).

### 3.4. Yo-yo problem

Yo-yo problem, или «проблема йо-йо» — когда у нас поток управления, как йо-йо, передаётся от потомка к предку, от предка к потомку, от потомка к предку и т.д. Это дальнейшее развитие «идеи» Call Super — давайте не только потомок будет вызывать методы предка, но и наоборот, предок будет вызывать виртуальные методы, которые будут работать у потомка, в которых, в свою очередь, будут вызываться методы предка, в которых будут тоже вызываться виртуальные методы и т.д. Такие ситуации не редки в продакшин-коде и являются следствиями благих намерений — красивой объектно ориентированной архитектуры, где всё общее поведение вынесено в предка, а потомки его только параметризуют своими действиями (кстати, «шаблонный метод», если потомок вызывает методы предка, тоже даёт проблему йо-йо).

Проблема это потому, что чтобы понять, как работает каждый класс в иерархии наследования, вам надо понять всю иерархию наследования одновременно, нет чёткого разделения ответственности между потомками и предками, и при отладке управление прыгает между файлами. В промышленных проектах встречались иерархии глубиной в пять уровней с проблемой йо-йо, отлаживать такой код было невозможно и пришлось всё переписать.

Если проблема йо-йо не очень запущенна, с ней можно бороться, просто перераспределив функциональность между предками и потомками, так, чтобы вызовы всегда были в каком-то одном направлении. В более тяжёлых случаях потребуется разделить иерархию наследования на несколько. На самом деле, вызовы предка — это обычно вызовы каких-то вспомогательных методов, которые нужны всем потомкам и находятся в предке просто для того, чтобы быть им доступными. Тогда можно вынести такие методы в отдельный класс. Если поведение таких методов тоже бывает разным в зависимости от чего-то, поможет паттерн «Мост».

Вообще, проблема йо-йо становится критичной, если иерархия наследования достаточно глубока, и вообще не возникает, если классы не наследуются друг от друга, а только реализуют интерфейсы (в частности поэтому во второй лекции говорилось, что наследование — это плохо). Так что хороший способ избежать проблемы йо-йо — избегать иерархий наследования глубины больше трёх, и вообще использовать наследование только как механизм полиморфных вызовов, используя композицию как способ переиспользовать общую функциональность.



(c) Wikipedia  
[https://commons.wikimedia.org/wiki/Category:Yo-yos#/media/File:Wooden\\_yo-yo.jpg](https://commons.wikimedia.org/wiki/Category:Yo-yos#/media/File:Wooden_yo-yo.jpg)

### 3.5. Busy waiting

Busy waiting, или «активное ожидание» — это ситуация, когда мы ожидаем наступления какого-то события в цикле, постоянно проверяя, не наступило ли оно. Программа как бы спрашивает, «произошло ли событие? а сейчас? а сейчас?» и так до тех пор, пока оно не произойдёт. Вариант этого антипаттерна — использование циклов для задержки: в конце концов, подождать  $N$  миллисекунд можно, выполнив  $M$  итераций цикла `for` с пустым телом. Плохо это из-за того, что процессор-то не знает, что цикл ожидания на самом деле ничего не делает, и старается исполнить его с максимальной возможной скоростью, полностью занимая одно из ядер, потребляя максимум электроэнергии и выделяя кучу тепла.

Тем не менее, это не всегда плохо, а во встроенных системах может быть вполне validным способом организации работы — например, для опроса датчиков. Там часто используются специализированные вычислительные устройства, которые работают на низкой частоте и потребляют очень мало электроэнергии, даже крутясь в бесконечном цикле, кроме того, они рассчитаны на то, что будут непрерывно работать. Процессоры же обычных настольных компьютеров или ноутбуков, напротив, спроектированы с учётом того, что будут применяться в основном в интерактивных приложениях, где 99% времени приходится ждать пользователя, так что работа на полную мощность для них скорее исключение, чем правило. Тем не менее, даже для обычных процессоров активное ожидание иногда применяется по делу — например, примитив синхронизации `SemaphoreSlim` в .NET перед тем, как заблокировать поток с помощью планировщика, ждёт в активном ожидании несколько (скорее несколько тысяч) циклов в надежде, что критическая область быстро освободится. Поскольку вызов планировщика трудозатратен, а критические области часто бывают короткими, это позволяет сделать синхронизацию в разы эффективней.

Тем не менее, активное ожидание очень часто происходит по ошибке, особенно в коде людей, только познакомившихся с принципами многопоточного программирования. Вообще, плохую многопоточную программу можно отличить от хорошей по характерному гулу системы охлаждения, когда процессор начинает греться, не выполняя полезной работы.

Бороться с этим антипаттерном можно с помощью планировщика, если он есть на целевой машине (во многих системах реального времени и тем более на голом железе планировщика нет). Любая нормальная ОС с планировщиком имеет системные вызовы, позволяющие усыпить поток до наступления какого-то события. Пример такого вызова — функция `select` в API Linux. Ещё примеры — `Event`-ы в Windows (и `conditional variable`-ы в Linux), примитивы синхронизации типа мьютексов, семафоров и мониторов. Если планировщика нет, то есть аппаратные возможности организации асинхронного исполнения — таймеры и аппаратные прерывания. Если и этого нет, то наверняка железо как раз заточено под активное ожидание и будет работать с ним вполне нормально (хотя лучше три раза перепроверить, конечно).

### 3.6. Error hiding

Error hiding, или «скрытие ошибки» — ситуация, когда сообщение об ошибке показывается без какой-либо диагностической информации или не показывается вовсе. Так происходит, естественно, из благих намерений не пугать пользователя, и в надежде, что

проблема некритична и если о ней не сообщать, то её и не заметят. Так часто делают продукты, нацеленные на массового покупателя — «Oops, something went wrong» в браузере является хорошим примером такого поведения. Антипаттерном это становится в том случае, когда диагностику оказывается вообще невозможно получить. Пример такого антипаттерна на «тактическом» уровне — это «проглатывание» исключений, когда исключение просто ловится пустым блоком `catch` и программа продолжает работать как ни в чём не бывало. Иногда это даже оправданно (там, где ошибки вполне ожидаемы, например, при работе с сетью), но чаще всего это приводит к некорректному состоянию программы, порче данных, боли и страданиям.

Бороться с этим антипаттерном довольно просто, но сложно психологически. Надо убедить себя, что пользователи не придут с вилами и факелами к офису вашей компании, если программа будет изредка падать, а наоборот, будут писать багрепорты, что позволит быстро выявить, локализовать и устранить проблемы с кодом. Хотя увлекаться не стоит, а то игровая индустрия часто принимает эту рекомендацию слишком буквально и новая игра часто пару месяцев просто не запускается у половины покупателей.

Есть известный принцип «Fail fast», рекомендующий заканчивать работу, как только выявлена проблема, и вообще в любой подозрительной ситуации. Это хорошо работает не только в программировании, но и, например, в стартапах — сначала вы делаете самую рисковую часть работы, и если с ней не получается, закрываете стартап и делаете новый. Чем быстрее вы «Fail», тем больше у вас будет попыток сделать что-нибудь хорошее. Также и в мире программного обеспечения — дайте программе упасть. Помогите программе упасть, добавив в код как можно больше различных проверок — проверки всех аргументов у всех паблик-методов, `assert`-ы для всех разумных инвариантов, не отключайте `assert`-ы в релизном коде. Чем быстрее код упадёт, тем больше шансов, что проблема будет выявлена на этапе тестирования, но если она всё-таки проскочит QA, тем быстрее пользователи о ней сообщат и вы сможете её поправить.

Ещё хорошее правило — логировать все интересные вещи, происходящие с программой. Как минимум, все исключения, но можно и внешние события, и крупные изменения состояния, чтобы понять по логу, что произошло.

### 3.7. Magic numbers, Magic strings

Magic numbers, Magic strings, или «магические числа», «магические строки» — это цепных два антипаттерна, похожих, но имеющих несколько разные последствия. Магические числа — это практически все встречающиеся в коде числовые литералы, за исключением 0 и иногда 1. Плохи они тем, что вы можете не иметь идей, почему это число именно такое, как написано в коде, и если надо что-то поменять, то какие именно числа в коде надо менять (число 100 всегда просто число 100, и вам предстоит угадать, это размер массива, который вы хотите увеличить вдвое, или желаемая скорость автомобиля на шоссе, которую лучше вдвое не увеличивать). Особенно магические числа хороши, если автор кода любезно посчитал значение какого-нибудь огромного выражения и вставил в код только его результат, а потом требования поменялись и какая-то константа в этом выражении изменилась (а само выражение потеряно и забыто).

Магические строки — это, соответственно, строковые литералы в коде. Их, внезапно, тоже быть не должно. Строки бывают двух крупных категорий — те, которые показываются пользователю, и те, которые используются только внутри кода. Строки, которые

показываются пользователю, надо локализовывать (даже если вы думаете, что вашим приложением будут пользоваться только англоговорящие или русскоговорящие люди, в реальной жизни так не бывает). Локализация требует массы усилий, а если о ней заранее не подумать, усилий требуется десятикратно больше. Не раз бывало, что крупные проекты занимались интернационализацией/локализацией более полугода, и только ей, не выпуская больше никаких обновлений и не разрабатывая новой функциональности.

Строки, которые пользователю не показываются, тоже не стоит оставлять в виде литералов, потому что можно в строке опечататься, она где-нибудь с чем-нибудь не сравнится и всё пойдёт не так. Выносите строки в константы, если они вам нужны, или старайтесь от таких строк вообще избавиться. Ещё частый пример таких строк — SQL-запросы или что-то такое. Опять же, в них легко опечататься, они не проверяются компилятором, поэтому использовать их в коде крайне не рекомендуется. Благо для генерации SQL-запросов есть куча хороших ORM<sup>1</sup>-систем — если в вашем коде есть строка с кодом на SQL, вы что-то делаете не так.

Бороться с магическими числами очень просто — вынесите их в именованные константы (или используйте enum-ы). С магическими строками сложнее — если они не показываются пользователю, то тоже константы. Если показываются, то надо использовать технологию переводов, поддержанную в вашем технологическом стеке. Например, в Microsoft-овских технологиях используется генерация именованных констант, которые потом можно использовать в коде, и использовать их разные значения в зависимости от выбранного языка (для этого в .NET есть система с локале-специфичными сборками). В C++/Qt есть очень удобный механизм Qt Linguist, где переводы — это XML-файлы, по которым генерируются по сути хеш-таблицы, отображающие исходные строки в переведённые, и вкомпилируются в саму программу. В любом случае, такие технологии обычно нацелены на переводчиков, не умеющих программировать (и иногда даже толком пользоваться компьютером), поэтому очень удобны. Есть даже веб-ресурсы для колаборативного перевода, ими тоже надо пользоваться (может, не сразу, но иметь их в виду).

## 4. Антипаттерны проектирования

### 4.1. God Object (The Blob)

Перейдём к более стратегическим антипаттернам, оказывающим влияние на всю архитектуру системы. Первый и наиболее известный из них — это God Object (или «божественный объект»). God Object — это когда в системе (или в компоненте) всем процессом вычислений управляет один класс, знает про все остальные и пользуется ими просто как хранилищем данных. Почему это плохо — потому что такой класс имеет тенденцию иметь впечатляющие размеры и чрезвычайную сложность, сопровождать его очень тяжело, а переиспользовать и вовсе невозможно. Фраза-детектор этого антипаттерна — когда коллега, вводящий вас в проект, говорит «This is the class that is really the heart of our architecture».

Причиной появления God Object в коде обычно является постепенное развитие proof-of-concept без рефакторинга. В индустриальной практике такое бывает очень часто — ещё до заключения договора технического специалиста просят проверить идею, он тратит пару

---

<sup>1</sup> Object-Relational Mapping, например, Entity Framework для .NET

дней и делает очень грубый набросок алгоритма, который, кажется, работает. Его показывают клиенту, чтобы получить фидбэк, клиент говорит «да, это именно то, что надо», подписывает контракт, и специалист радостно думает, что пора начинать писать нормально. Приходит к менеджеру, а тот ему говорит, что надо в прототип ещё такую и такую фигу добавить. Специалист пытается возразить, что прототип вообще не предназначен для продакшн-окружения и его надо переписать, но менеджер отвечает, что клиент доволен, а это главное, надо быть клиенто-ориентированным, и что ему на самом деле будет трудно объяснить клиенту, почему на первом демо всё работало, а теперь надо ждать полгода, чтобы получить то же самое. Ну и начинается вечная гонка за дедлайнами, где для рефакторинга времени никогда не остаётся.

Ещё God Object иногда появляется у начинающих программистов из-за привычки к структурному программированию или просто привычки писать на скриптовых языках, не задумываясь об архитектуре. Или это может быть честная архитектурная ошибка, вызванная плохим разделением системы на компоненты и ошибкой в назначении обязанностей.

В любом случае, божественные объекты обычно сурово нарушают принцип единственности ответственности, представляя собой хаотичное объединение разных ответственостей в один класс. Часто это выражается в большом количестве полей и методов — но этим свойством обладает ещё антипаттерн Swiss Army Knife, так что по одному этому критерию непонятно: возможно, вы имеете дело не с God Object, а с ним. Обычно «большое количество» — это более 60, хотя, конечно, более важный признак — это принцип единственности ответственности. Например, типичный класс String вполне может иметь более сотни методов, но они все там по делу, работают со строками и God Object-ом этот класс не является.

Ещё важное исключение — это классы-обёртки над legacy-кодом. У них может быть много несвязных методов, как у настоящего God Object, но просто потому, что legacy-код плохой, тогда как сами эти методы — простые односторонники. Такие штуки можно оставить как есть, потому что обычно с legacy-кодом всё равно ничего не сделать, и его не надо поддерживать.

Вот что можно сделать, если у вас завёлся God Object.

- Передать больше ответственности классам-данным. Они ведь наверняка могут проявить некоторую самостоятельность и делать часть работы сами. Затем можно посмотреть, что ещё можно из божественного объекта передать классам-данным, и продолжать в таком духе, пока не получится нормальная архитектура.
- Разделить методы класса на группы, соответствующие контрактам, выполняемым God Object-ом. Возможно, уже существуют классы, в которые можно перенести эти группы, например, те же классы-данные, либо какие-то ещё классы системы. Возможно, нет, тогда их надо создать, так, чтобы каждый класс отвечал за что-то одно (принцип единственности ответственности всегда должен направлять рефакторинг).
- Убрать непрямые зависимости: если объекты А и В лежат внутри объекта G, может так случиться, что А может быть в В, а В — в G. Это позволит God Object-у G меньше знать, что всегда позитивно.

## 4.2. Swiss Army Knife

Swiss Army Knife, или «швейцарский нож» — это антипаттерн, очень похожий на God Object, но не пытающийся управлять всеми вычислениями. Это класс с очень сложным интерфейсом, пытающийся делать всё, что в принципе может кому-то понадобиться. Такие классы часто появляются в библиотеках, разработчики которых пытаются сделать свои абстракции максимально гибкими, чтобы они могли применяться в самых разных контекстах. Казалось бы, гибкость — это хорошее свойство, но получающимися абстракциями пользоваться очень тяжело, требуется передавать десяток параметров и конструировать десяток вспомогательных объектов, чтобы сделать даже самое простое действие.

На самом деле, это явление имеет важные причины с точки зрения архитектуры вообще. Абстракция как таковая должна предоставлять простой интерфейс к потенциально сложной реализации. Инкапсуляция защищает реализацию от внешнего мира, что хорошо не только для абстракции, но и для внешнего мира — ему не надо знать, как абстракция устроена. Идеальная абстракция — класс с одним методом «сделать зашибись» без параметров, который решает задачу пользователя, у такой абстракции и инкапсуляция идеальна. Но такой метод может решать ровно одну задачу, так что не очень гибок. Поэтому ради гибкости приходится жертвовать простотой использования, интерфейс абстракции чем более гибок, тем более сложен, и если продолжать этот процесс до бесконечности, то можно предоставить просто интерпретатор какого-либо тьюринг-полного языка, на котором можно сделать что угодно, но никакой радости пользователю это не принесёт. Хорошая абстракция должна балансировать гибкость и сложность (инкапсулировать сложность от пользователя), и Swiss Army Knife — это ситуация, когда баланс слишком смещён в сторону гибкости.

В отличие от God Object, Swiss Army Knife не пытается монополизировать управление в программе, так что швейцарских ножей может быть много, тогда как God Object, как правило, один. Ещё одно важное отличие — God Object может иметь очень простой public-интерфейс и делать всю работу в private-методах (ему не нужен развитый внешний интерфейс, потому что он всё делает сам); Swiss Army Knife всегда имеет много public-методов.

## 4.3. Lava Flow

Название следующего антипаттерна не то чтобы на слуху, но он встречается в подавляющем большинстве проектов — Lava Flow или «поток лавы». Суть антипаттерна в том, что эксперименты, быстрый поиск решения и прототипирование выполняются прямо в основном коде, после чего, когда решение найдено, остатки этих экспериментов не вычищаются должным образом (из-за вполне нормальной для всех проектов спешки) и застывают в коде навсегда — потому что трогать их страшно, а вдруг это кому-то всё-таки нужно и если тронуть, то сломается. Фраза-детектор антипаттерна: «Oh that! Well Ray and Emil (they're no longer with the company) wrote that routine back when Jim (who left last month) was trying a workaround for Irene's input processing code (she's in another department now, too). I don't think it's used anywhere now, but I'm not really sure. Irene didn't really document it very clearly, so we figured we would just leave well enough alone for now. After all, the bloomin' thing works doesn't it?!». Думаю, что все когда-то что-то такое слышали.

В более общем виде Lava Flow представляет собой любой незакрытый технический

долг, постепенно накапливающийся в системе, избавиться от которого очень сложно, даже если это просто мёртвый код. Появляется он, как правило, как быстрые фиксы к каким-то насущным проблемам (знаменитое «Завтра релиз, просто сделай, чтобы оно работало»), которые потом никогда не заменяются нормальным решением (а зачем? Оно же работает! Ну и потом, «завтра релиз» заменяется на «релиз через две недели» и задач на месяц). Усугубляется всё естественной потерей знаний об этих костылях за счёт ухода людей из команды и невозможности для оставшихся в деталях изучить все закоулки системы. Поэтому обычно даже если человек видит спорное техническое решение в коде, он предпочитает его не трогать (*«It doesn't really cause any harm, and might actually be critical, and we just don't have time to mess with it.»*).

Признаками запущенного Lava Flow является закомментированный код, большие методы без комментариев, непонятные public-методы с загадочными именами и не менее загадочными параметрами. Закомментированный код особо опасен, потому что он как бы есть, но компилятор его не проверяет, система вокруг него меняется, так что он очень быстро перестаёт быть актуальным, и если его раскомментировать, работать он наверняка не будет. Закомментированный код надо уничтожать сразу же.

Lava Flow проще предупредить (насколько это возможно), чем от него потом избавиться. Все эксперименты, concept proof, варианты решения, «быстро попробовать кое-что» и т.д. и т.п. следует всегда делать в отдельных ветках, которые потом безжалостно прибивают, получив нужное знание. Не пишите код, который попадёт в production, до тех пор, пока вы не знаете, что делаете, и настаивайте на том, что прототип не должен поставляться клиенту. Нормальный порядок разработки технологически рискового проекта должен начинаться с фазы R&D, где вы играете с технологиями и алгоритмами в отдельном изолированном проекте, а когда готовы — выкидываете его, разрабатываете архитектуру уже начисто, и пишете код с нуля или практически с нуля.

Если всё-таки Lava Flow уже в проекте, причём серьёзно мешает разработке, требуются весьма радикальные меры. Если удаётся его победить небольшими рефакторингами, то хорошо — но помните, что постепенная борьба с техническим долгом будет неизбежно приводить к новым багам, которые нельзя чинить новыми костылями. Кроме того, убиранье одного простого костыля на пару строчек приводит зачастую к перепроектированию большого куска приложения, чтобы «сделать нормально». Если такого перепроектирования требуется выполнять много, то имеет смысл провести полноценный архитектурный реинжиниринг системы — проанализировать архитектуру как она есть и разработать архитектуру как она должна быть, плюс план наиболее безболезненного перехода от одного к другому. Это может быть очень долгий и сложный процесс, который, к тому же, приводит к полной остановке разработки — нет смысла реализовывать новую функциональность илификсить баги, если система всё равно будет практически переписана в ближайшем будущем. Бывали ситуации, когда такой процесс занимал в индустриальных проектах до полугода сконцентрированных усилий.

Как вариант всегда стоит рассматривать идею «выкинуть всё и написать заново». Несмотря на кажущуюся апокалиптичность этого решения, написание новой системы с опытом и знаниями, полученными при разработке старой, займёт в разы меньше времени. Поэтому так довольно часто поступают и в больших проектах — Windows была как минимум один раз полностью переписана, .NET был полностью переписан, таких примеров (успешных!) можно привести ещё много.

## 4.4. Functional Decomposition

Functional Decomposition, или «функциональная декомпозиция» — антипаттерн, заключающийся в проектировании объектно-ориентированной системы в функциональном стиле. Характеризуется архитектурой, построенной называющих друг друга функциях, которые постепенно декомпозируют и решают задачу. Типичны классы с функциональными названиями и одним методом, обилие статических классов или один большой класс с одним public-методом и кучей private-методов. Фраза-детектор: «This is our main routine, here in the class called LISTENER».

Обратите внимание, что это антипаттерн только в том случае, если вы программируете на объектно-ориентированном языке. Программы на Pascal, C, Ada, не говоря уже о функциональных языках наподобие Haskell, прекрасно пишутся в процедурном стиле и столь же прекрасно работают, там это не антипаттерн. Более того, думаю, что можно считать антипаттерном попытку объектно-ориентированно программировать на процедурных языках (хотя вот симуляция ООП на С обычно считается чем-то хорошим). Кроме того, даже в объектно-ориентированных языках выполнять первые этапы декомпозиции относительно потока выполнения может быть хорошей идеей, так, например, устроен архитектурный стиль Pipes and Filters. Плохо применять процедурное программирование на реализационном уровне.

Если вдруг такая проблема возникла, что можно сделать:

- вернуться к требованиям, попробовать построить объектно-ориентированную модель предметной области «с нуля»;
- потом надо отобразить эту модель на уже существующий код, в духе «вот этот метод этого объекта у нас реализуется такой-то и такой-то функцией». На этом этапе не советуют кидаться рефакторить код, потому что запутаешься, цель — задокументировать и привязать к требованиям существующий код, чтобы потом понять, что с ним делать.
- Собственно рефакторинг можно начинать с лёгких целей — классы с одним методом можно прибить, переместив код из них в другие классы.
- Затем кластеризовать то, что получилось, в группы методов, которые соответствуют классам из предметной области, и сделать их полноценными классами.
- Если в классе нет состояния, можно сделать его статическим классом, в надежде, что состояние у него потом заведётся в процессе рефакторинга само собой. Если нет, можно перенести код в какой-нибудь из настоящих классов, либо решить для себя, что это на самом деле класс-сервис, у которого состояния быть и не должно. Так бывает, например, когда класс выполняет операции над двумя другими классами и каждый из этих классов играет равную роль в этой операции (так что её нельзя по смыслу перенести в один из этих классов).
- Есть ещё один радикальный метод борьбы с запущенной функциональной декомпозицией — свалит весь код в одну кучу, сделав God Object. А с God Object мы уже знаем как бороться.

## 4.5. Golden Hammer

Наконец, самый популярный в программистском сообществе антипаттерн — *Golden Hammer*, «золотой молоток». Это ситуация, при которой все проблемы пытаются решить одним инструментом. Например, вы любите C++ и вам кажется хорошей идеей разрабатывать на C++ веб-сервисы. Или вы любите Python и вам кажется хорошей идеей разрабатывать на Python настольные приложения с богатым пользовательским интерфейсом. Более запущенные ситуации отражают эти фразы-детекторы: «*Our database is our architecture*», «*Maybe we shouldn't have used Excel macros for this job after all*».

На самом деле, «золотой молоток» — это вовсе не появление глупости или недалёкости, а вполне обусловленная экономически модель поведения. Если вы потратили 20 лет на то, чтобы выучить наизусть стандарт C++ и три его самых распространённых диалекта, вы, естественно, не хотите переходить на Java, где вы новичок. Вообще, рост опыта во владении конкретным продуктом, технологией или стеком пропорционален желанию использовать их везде, где только можно. Ваши навыки — ваше конкурентное преимущество на рынке труда, кроме того, человек, средне владеющий десятком технологий, ценится обычно меньше, чем человек, в совершенстве владеющий одной технологией. Ситуация усугубляется желанием вендоров посадить на свою технологию как можно больше разработчиков, для чего сделать свою технологию универсальной (вспомним Node.js).

Собственно, антипаттерн — это не владеть выбранным технологическим стеком (это всегда хорошо), а нежелание (или даже активное сопротивление) посмотреть по сторонам. Пересесть на новый язык программирования и стек технологий у опытного программиста занимает не больше недели, ещё через пару месяцев он станет вполне продуктивным, а через полгода будет владеть технологией в совершенстве. Поэтому, кстати, в СПбГУ нет курсов по Java, зато есть курсы по архитектуре, многопоточному программированию, компьютерным сетям и т.п. В общем, чтобы не оказаться в ловушке золотого молотка, перед тем, как выбирать технологию реализации очередного проекта, надо погуглить, на чём обычно такие проекты делаются.

Важная проблема золотого молотка (почему он, собственно, антипаттерн, помимо потерянной эффективности разработки) — это то, что используемые технологии сильно влияют на создаваемое решение даже в плане требований. Например, если поддержка веб-сервисов не сильная сторона C++, вы будете пытаться навязать клиенту приложение на голых сокетах, даже если ему нужно интегрироваться с другими системами по SOAP. Даже подсознательно все требования будут проходить через фильтр «а не слишком ли это сложно сделать на технологии X?», и в итоге клиент получит не тот продукт, что ему нужен, а тот, который не очень сложно запрограммировать на вашей любимой технологии.

Как бороться с «золотым молотком» — прежде всего, постараться убедить себя, что вы не Java-программист, C++-программист, Ассемблер-программист, а просто программист. Что если у вашей команды нет экспертизы в технологии X, это повод таковую экспертизу получить. Как себя в чём-либо убедить, вопрос сложный, но сделать это необходимо. Язык и технология — это инструменты решения задачи, а не определение вашей личности.

Может помочь обучение. Например, внутренкорпоративные семинары, где опытные разработчики рассказывают другим командам про свои любимые технологии и оказывают поддержку коллегам в их внедрении. Ещё очень хорошо открывают глаза на вещи поездки на конференции (технические или даже научные), митапы. Как руководитель организации не стесняйтесь тратить на это деньги, как сотрудник — не стесняйтесь пользоваться

предложениями от начальства.

Архитектура тоже может помочь в борьбе с «золотым молотком». Если проектировать систему как набор слабо связанных заменяемых компонентов, можно добиться того, чтобы каждый компонент можно было разрабатывать на своём технологическом стеке. Особен-но полезны в этом плане микросервисная архитектура и сервисо-ориентированная архи-тектура вообще — каждый сервис запускается в своём процессе, так что пока он следует стандартам на коммуникацию (SOAP, REST), он может быть написан на чём угодно. В Facebook, кажется, разработчикам разрешено писать на любом языке, который им больше нравится, именно потому, что архитектура микросервисная, микросервис можно с нуля переписать за две недели, и каждый имеет свой жизненный цикл, лишь бы с ним могли нормально общаться другие.

Если микросервисы для вас неприменимы, то вполне могут быть индустриальные стан-дарты по разделению на компоненты и коммуникации между компонентами (так называ-емые reference architectures) или широко распространённые middleware, например ROS<sup>2</sup>. Также стоит помнить про «кросс-технологические» инструменты коммуникации, напри-мер, Protobuf/gRPC, Apache Thrift и т.п.

И главное — не надо бояться новых технологий, новых языков и т.д. Квалифициро-ванный программист может хорошо программировать на чём угодно, потратив пару дней на то, чтобы почитать про это что угодно немного.

## 5. Антипаттерны архитектуры

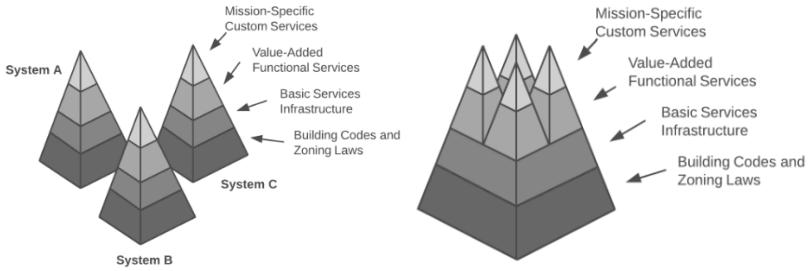
Антипаттерны архитектуры — это антипаттерны ещё более стратегические, чем рас-смотренные ранее. Они зачастую затрагивают решения на уровне архитектур приложений целой организации и больше ошибки менеджмента, чем ошибки технических специали-стов. Тем не менее, знать и избегать их неплохо бы каждому.

### 5.1. Stovepipe Enterprise (Island of Automation)

Stovepipe Enterprise (Island of Automation), «Организация в стиле печной трубы», «Ост-ров автоматизации» — ситуация, когда в организации есть несколько проектов, которые никак не помогают друг другу. Такое в индустриальных компаниях встречается до-вольно часто, иногда из-за кажущейся невозможности организовать переиспользование кода на разных языках и технологиях, иногда по кажущимся соображениям авторского права. В любом случае, идеи, знания, типовые архитектуры переиспользовать можно, и хорошие организации к этому всячески стремятся. «Печная труба» тут символизирует ста-рую печную трубу, которую часто приходится латать, и её латают чем попало, превращая в хаотичное нагромождение заплаток. Суть дела поясняет картинка:

---

<sup>2</sup> Robot Operating System, которая вовсе не операционная система, а скорее фреймворк для разработки распределённых приложений в области робототехники



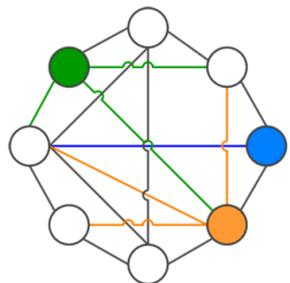
(C) SourceMaking.com  
<https://sourcemaking.com/antipatterns/stovepipe-enterprise>

Как бороться с такой ситуацией:

- использовать существующие промышленные стандарты. Например, если вся организация работает в сфере робототехники, стандарт де-факто по интеграции компонентов там ROS, так что достаточно мелкие компоненты из разных проектов вполне могут оказаться совместимы и переиспользуемы, а опыт использования ROS как платформы — переиспользуем.
- Фиксировать технологии и типовую архитектуру на уровне предприятия. Это идёт несколько вразрез с тем, что говорилось про «золотой молоток», но если вы берёте только проекты на .NET, сделайте себе репозиторий на всю организацию, положите туда библиотеку переиспользуемых компонентов, настройте NuGet-сервер, создайте шаблоны типовых приложений и т.д.
- Вообще, создать и поддерживать инфраструктуру переиспользования. Она может варьироваться от просто корпоративного репозитория с полезным кодом до in-house генераторов типовых приложений или даже предметно-ориентированных визуальных языков (см. опыт Nokia из <https://www.metacase.com/cases/nokia.html>). Начальные инвестиции могут быть высоки и это, безусловно, отпугивает, но результирующие преимущества могут быть впечатляющими (тот же опыт Nokia как экстремальный пример, когда in-house-технология позволила нанять неквалифицированных специалистов и дать им возможность производить по одному несложному мобильному приложению в день, что вручную было вообще недосягаемо).

## 5.2. Stovepipe System

Stovepipe System, «Система в стиле печной трубы» — уточнение Stovepipe Enterprise для отдельной системы, состоящей из компонентов. Характеризуется отсутствием единого стандарта по взаимодействию между подсистемами и интеграцией подсистем по принципу «как получится» (*«ad-hoc»*). В этом случае отработанные решения по взаимодействию между компонентами опять-таки не могут помочь друг другу, добавление нового компонента или изменение связей между существующими — большая работа, а тестирование такой системы — то ещё удовольствие.



(C) SourceMaking.com  
[https://sourcemaking.com/antipatterns/stovepipe-  
system](https://sourcemaking.com/antipatterns/stovepipe-system)

Как с этим бороться — как обычно, введением внутренних стандартов. Тут могут помочь и уже существующие стандарты (хороший пример — тот же ROS, или REST-сервисы), и использование известных паттернов интеграции — например, Enterprise Service Bus. Более «тактическое» решение — выделение единого интерфейса для подсистем и реализация всего взаимодействия в терминах этого интерфейса (так обычно устроены pluginные системы, например).

### 5.3. Vendor Lock-In

Антитриммер Vendor Lock-In, или «зависимость от поставщика» — ситуация, когда архитектура вашей системы или её реализация зависит от третьестороннего коммерческого решения, которое вы не можете контролировать. Казалось бы, почему бы не разрабатывать приложение по поддержке электронного обучения HwProj на Ruby, молодой и перспективной технологии? Потому что HwProj используется через семь лет после своего запуска, а Ruby нет — специалистов, способных поддерживать HwProj, не найти, баги не фиксятся, новые фичи не реализовать. Это весьма типичная ситуация для приложений, у многих из них ужасно долгий жизненный цикл, длищийся десятки лет. Часто приложения надолго переживают технологии, с помощью которых были разработаны. Хороший пример тому — банковские системы, написанные на COBOL в 80-х годах двадцатого века. Они используются до сих пор, и их слишком опасно и дорого менять, поэтому будут использоваться и дальше, а COBOL помер давно. А сопровождать такие системы надо.

Собственно, долгий жизненный цикл приложений — не антипаттерн, а суровая реальность. Антипаттерн — завязываться без нужды на вендора, особенно если нет оснований для уверенности, что он будет поддерживать свою технологию на протяжении ещё хотя бы пары десятилетий. Особенно если эта технология навязывает архитектуру всему приложению и неотделима от полезного кода (например, это платформа, на которой приложение разрабатывается). Если речь идёт про Java, .NET или даже Qt, вряд ли стоит от них отказываться в пользу самодельного компилятора (хотя в критических приложениях типа ОС вооружённых сил — стоит ещё как!), но если вы пишете приложение на долгие годы вперёд и делаете это на новом хипстерском js-фреймворке известного хипстерского стартапа

«Рога и Копыта», то надо принять меры на случай исчезновения фреймворка с рынка.

Какие это могут быть меры:

- опять-таки, архитектурная подготовка к изменениям — микросервисы, просто сервисы, изолированные компоненты;
- избегать сильной привязки к подозрительным третьесторонним компонентам — программировать не «на», а «с помощью»;
  - интересная тактика по этому поводу используется в некоторых промышленных проектах — выкладывать в свой репозиторий и компоненту, от которой мы зависим, причём даже в бинарном виде, хоть это и считается табу для систем контроля версий — пока наш репозиторий жив, зависимости тоже живы;
  - более грамотные люди используют Docker-образы для подобных целей;
- использовать изоляционный слой — тонкую прослойку кода между вашей системой и третьесторонним компонентом. Компонент перестали выпускать и вообще убрали из интернета — не проблема, выкидываем его и заменяем на другой, или пишем свою реализацию, при этом существующий код, кроме изоляционного слоя, менять не надо.
- Не бояться реинжиниринга. Если сама платформа, на которой вы писали, померла, может потребоваться выполнить перенос приложения на новую платформу, с использованием методов реинжиниринга типа извлечения бизнес-правил. Это дорого и сложно, но дешевле переписывания с нуля (хотя и само переписывание с нуля в разы дешевле разработки в незнакомой предметной области).

## 5.4. Architecture By Implication

Architecture By Implication, или «неявная архитектура» — это антипаттерн архитектуры, отрицающий необходимость явной спецификации архитектуры в случае, если уже есть опыт создания подобных приложений. Фразы-детекторы: «We've done systems like this before!» и «“There is no risk; we know what we're doing!». Казалось бы, это логично, если опыт есть, есть и типовая архитектура с предыдущего проекта, и технологии, и даже наработки, зачем ещё раз делать одно и то же?

Затем, что требования всё-таки могут различаться, и задачи, кажущиеся похожими, могут содержать в себе скрытые риски, которые при отсутствии явной разработки архитектуры могут остаться незамеченными. Возможно, вы настолько уверены, что задача похожа на предыдущую, что даже не понимаете, что заказчик хочет совсем другого. Плюс к тому, даже если всё хорошо, есть риск «золотого молотка», когда вы снова достаёте проверенную дедовскую технологию, которой сделали уже 20 проектов, и не хотите погуглить новые вещи, которые вашу задачу позволяют решить в разы быстрее и лучше.

Обратите внимание, переиспользовать архитектурные решения с предыдущих проектов никто не запрещает, причём выработка типовых архитектур и локальной библиотеки паттернов даже очень приветствуется, антипаттерн тут — отсутствие явной архитектурной документации вообще.

Как с этим бороться — требовать явного описания архитектуры системы всегда. Можно использовать стандарты Software Design Description (IEEE 1016) или Architecture

Description (ISO/IEC 42010), можно (что чаще бывает) выработать свой какой-то внутренний стандарт, но некий диздок должен быть. Можно подойти к этому ещё более структурно и использовать какой-либо архитектурный фреймворк (или просто методологию), где написано, что, когда и как надо создавать, чтобы архитектура считалась нормально описанной. Главное, не оставлять это дело на откуп примерному пониманию программистов.

## 5.5. Design By Committee

Design By Committee, «проектирование комитетом» — попытка принимать архитектурные решения простым большинством голосов («*A camel is a horse designed by a committee*»). Обсуждать архитектуру большими и шумными компаниями хорошо и правильно, но решения должны либо приниматься единогласно, либо должен быть один человек, который их принимает и отвечает за них. Иначе попытка включить в дизайн идеи, соображения и индивидуальные предпочтения каждого приводит к сложной, объёмной и внутренне противоречивой архитектуре. Хорошие системы вообще дизайняются одним человеком, либо командой, где есть один главный архитектор, у которого есть видение и идея системы, и помощники, которые либо проектируют отдельные компоненты, либо помогают уточнить дизайн и оформить документацию. Особо опасен этот антипаттерн, когда комитет — это консорциум из сотни крупных компаний, каждая из которых лоббирует свои интересы и пытается отрызть долю рынка, «правильно» влияя на проект.

Что можно сделать, чтобы избежать этой ситуации — во-первых, ограничить размер команды. Идеал — 4 человека, почти идеал — 6-7 человек (команду, которую «можно накормить двумя пиццами»). У Бруска в книжке «Мифический человеко-месяц» была интересная идея, что в команде собственно программирует систему только один человек, остальные занимаются вспомогательными задачами (там это называлось «бригада главного хирурга»). Такая модель не прижилась, но в любой команде всё равно можно добиться разделения ролей, и надо его строго придерживаться. Должен быть явный главный архитектор, product owner, разработчики, эксперты и т.д. Это всё может быть одно лицо или целые группы, но роли должны быть чётко определены и лицо, принимающее решения, должно быть одно.

# Лекция 10: Архитектурные стили

Юрий Литвинов

yurii.litvinov@gmail.com

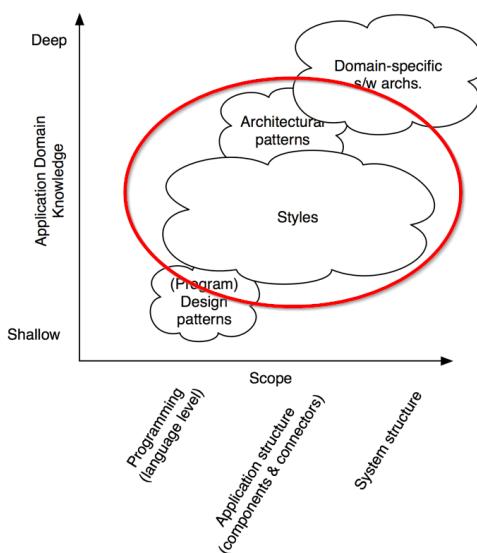
## 1. Архитектурные шаблоны и стили

Эта лекция — пожалуй, самая «архитектурная» в этом курсе, на ней пойдёт речь о наиболее известных архитектурных стилях. Вообще, архитектурный стиль — это набор решений, которые:

1. применимы в выбранном контексте разработки,
2. задают ограничения на принимаемые архитектурные решения, специфичные для определённых систем в этом контексте,
3. приводят к желаемым положительным качествам получаемой системы.

Есть ещё архитектурные шаблоны — это именованный набор ключевых проектных решений по эффективной организации подсистем, применимых для повторяемых технических задач проектирования в различных контекстах и предметных областях.

Определения довольно размытые, но суть дела поясняет рисунок:



Паттерны проектирования, которые мобсуждались в предыдущих лекциях — самые «тактические» элементы архитектуры. Они никак не привязаны к предметной области и появляются на уровне реализации небольших подсистем или даже конкретных классов. Архитектурные стили, о которых в основном пойдёт речь сегодня, применимы уже не для всех проектов вообще, а в предпочтительных для каждого стиля предметных областях — одни стили хорошо работают во встроенных системах, другие — сетевых приложениях, третьи — в информационных системах (отсюда «применимы в выбранном контексте разработки» из определения). И решения, диктуемые стилями, применяются не на уровне конкретных классов, а на уровне подсистем или даже целой системы.

Архитектурные шаблоны — это более специализированная вещь, чем стили, и несколько более «тактическая» (хотя и не настолько, как паттерны). Архитектурные шаблоны диктуют типовые решения для типовых задач, например, организация системы в виде тройки Sense-Compute-Control в робототехнике, или Model-View-Controller в пользовательских интерфейсах. Model-View-Controller не претендует на то, чтобы диктовать архитектуру всего приложения, и тем отличается от архитектурных стилей — обычно MVC лишь вершина айсберга, ответственная за общение с пользователем, а настоящая Архитектура начинается на уровне бизнес-логики, с которым работает Model.

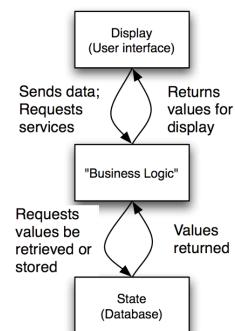
Ещё выше по масштабности и глубже по погружению в предметную область находятся предметно-ориентированные архитектуры или Reference Architectures (например, Space AVionics Open Interface aRchitecture от ESA, Connected Vehicle Reference Implementation Architecture и т.д. и т.п.). В каждой предметной области они свои и, как правило, даже диктуются стандартами. Рассматривать их в этом курсе мы не будем в силу их чрезмерной специфичности.

## 2. Архитектурные шаблоны

Вот несколько примеров архитектурных шаблонов. Они тоже специфичны для предметных областей, но некоторые часто встречающиеся достойны рассмотрения.

### 2.1. State-Logic-Display

State-Logic-Display, также известный как «трёхзвенная архитектура», часто рассматривается как архитектурный стиль, часто — как архитектурный шаблон, так что вообще разделение на архитектурные шаблоны и архитектурные стили весьма условно. Многие приложения могут целиком быть реализованы по трёхзвенной схеме, причём ни в одном из этих звеньев не будет содержательной архитектурной сложности, тогда это что-то вроде стиля. Если трёхвенка — это только способ решения одной из задач, то это архитектурный шаблон. Вообще, трёхвенка предполагает разделение приложения на часть, отвечающую за представление и взаимодействие с пользователем (и ничего больше), бизнес-логику и слой хранения данных.



Наличие чёткого разделения ответственности между слоями делает каждый из них управляемым, а функциональность — переиспользуемой. Например, пользовательский интерфейс легко поменять, сделав вместо десктопного приложения мобильное, не меняя бизнес-логики. В больших приложениях все три части могут быть физически размещены на разных машинах, причём пользовательских клиентов может быть много и они все работают со слоем бизнес-логики, который один. Или можно иметь несколько серверов с бизнес-логикой, лишь бы они смотрели на одну базу данных. Которая тоже может быть распределённой, так что это очень хорошо оказывается на масштабировании.

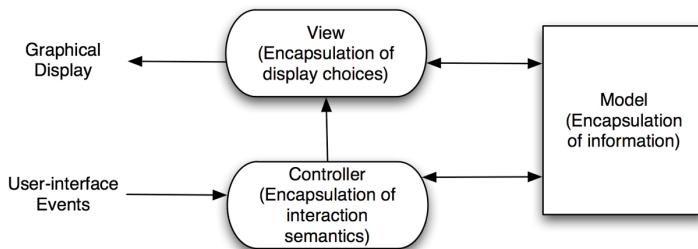
Главные архитектурные ограничения — клиент не знает ничего о БД и не может с ней напрямую взаимодействовать, бизнес-логика сама не пытается хранить информацию и не содержит ничего, что касается взаимодействия с пользователем, база данных не пытается делать ничего нетривиального с данными.

Приложения, устроенные таким образом — это подавляющее большинство веб-приложений и информационных систем (которые чаще всего сами веб-приложения), многопользовательские игры. В них клиент может быть очень продвинутым и очень сложным, но он не вправе заниматься реализацией игровой логики и не может хранить данные, кроме тех, что относятся к взаимодействию с пользователем (например, его логин).

## 2.2. Model-View-Controller

Model-View-Controller — это архитектурный шаблон, который иногда классифицируют как паттерн проектирования: собственно, в книжке про паттерны Эриха Гамма со товарищи с него и начинается изложение. Так что даже разделение на паттерны и архитектурные шаблоны тоже весьма условно. Кажется, что это всё-таки что-то большее, чем паттерн, поскольку он не предписывает наличия конкретных классов, да и сам часто реализуется через некоторые паттерны (например, «Наблюдатель», «Команда»).

Архитектурный шаблон устроен следующим образом:



© N. Medvidovic

View отвечает за отображение данных пользователю и только за это. Пользовательский ввод поступает в Controller, ответственность которого — обеспечить логику взаимодействия с пользователем и при необходимости управлять для этого View. Model — компонент, хранящий в себе все данные и, как правило, включающий в себя также и бизнес-логику приложения. Контроллер обрабатывает пользовательский ввод, сообщает о требуемых действиях модели, модель их выполняет, меняет данные и рассыпает нотификацию о том, что в ней что-то изменилось. Эту нотификацию получает представление, читает информацию из модели и обновляет себя.

Архитектурные ограничения: представление может только читать из модели, команда модели может отдавать только контроллер. Сигнал об обновлениях модель рассыпает сама, что позволяет иметь несколько разных представлений, отображающих одну модель, которые будут синхронно обновляться. Контроллер — единственное место системы, через которое проходят все команды пользователя.

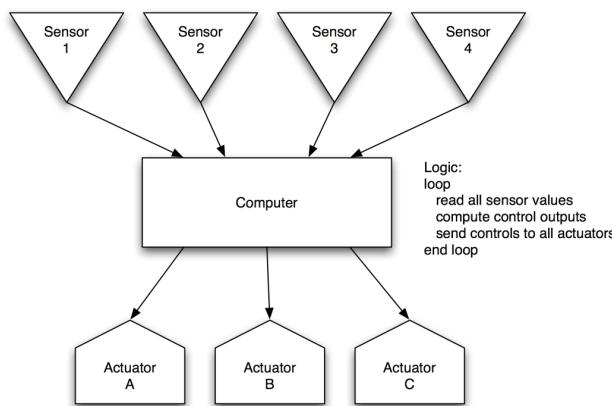
Чем это хорошо — так же, как в трёхзвенке, имеется чёткое разделение ответственности между компонентами и возможность выкинуть представление (или контроллер) и заменить на новое. Кроме того, контроллер — это естественное место для реализации функциональности Undo/Redo и обычно реализуется с помощью паттерна «Команда».

Model-View-Controller отличается от трёхзвенки тем, что не специфицирует, кому где хранить данные (поэтому применим в приложениях, которым данные особо хранить не надо), но требует наличия выделенного элемента, отвечающего за обработку пользовательского ввода (в трёхзвенке это делал Display, вместе с выводом информации).

Типичные приложения, построенные по такому принципу — это большинство десктопных приложений с развитым пользовательским интерфейсом. Как правило, впрочем, Model-View-Controller является лишь малой частью их архитектуры, а вся логика и вся архитектурная сложность скрыта под Model.

## 2.3. Sense-Compute-Control

Sense-Compute-Control — архитектурный шаблон, применяющийся прежде всего в робототехнике и схожих областях (например, «умных домах»). Он предполагает разделение работы системы на три фазы — снятие показания с датчиков, вычисление управляющего воздействия, посылка управляющего воздействия на актуаторы:



© N. Medvidovic

При этом вся система работает в бесконечном цикле, попеременно выполняя эти три фазы.

Очень простые системы (например, робот, едущий по датчику расстояния вдоль стены) на самом деле больше ничего интересного не содержат, но в общем случае фаза

Compute может включать в себя построение и обновление модели внешнего мира (например, целый большой и страшный SLAM<sup>1</sup>), кучу сложной логики по определению поведения исходя из текущей модели мира и новых данных сенсоров.

Хорошо это тем, что чётко определяет архитектуру системы и для простых систем фактически решает все архитектурные вопросы. Для сложных систем это может быть только самый внешний каркас процесса работы, но поэтому это и не архитектурный стиль, а всего лишь шаблон.

### 3. Архитектурные стили

Архитектурные стили менее специализированы, чем архитектурные шаблоны. Так же, как и шаблоны, стили имеют имя и набор известных свойств, которые позволяют выбрать стиль, подходящий под решение задачи. Архитектурные стили в разработке ПО часто сравнивают с архитектурными стилями в архитектуре. Так же, как в архитектуре, стили могут быть совсем разными, при этом решать схожие задачи и, в конечном итоге, быть отражением вкусовых предпочтений архитектора.



© N. Medvidovic

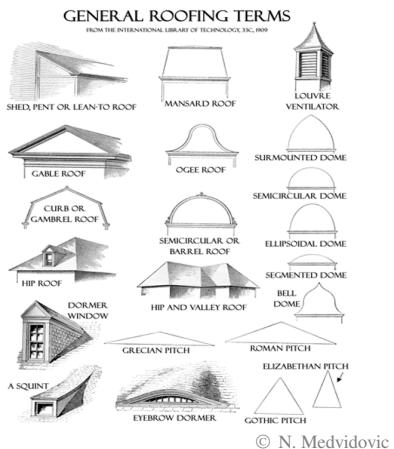
Однако не все архитектурные стили хороши для всех ситуаций. Пример, который приводил N. Medvidovic в своей лекции — в Калифорнии делают лёгкие каркасные дома из деревянного бруса, а в Сербии, откуда он родом — большие дома из камня, передающиеся из поколения в поколение. Дело в том, что в Калифорнии высока сейсмическая активность, которая деревянным домам не страшна, а каменные бы и пары лет неостояли. В Сербии землетрясений не бывает, а камня в достатке.

<sup>1</sup> Simultaneous Localization And Mapping

При этом одна система вполне может включать в себя несколько разных архитектурных стилей (опять-таки, по аналогии со строительством, где есть, например, разные архитектурные стили крыши). Для каждой подсистемы стиль может быть свой. Например, система может быть организована в слоистом стиле, но слой бизнес-логики реализован в стиле Pipes and Filters, а слой пользовательского интерфейса — в стиле «ядро и плагины». Тем не менее, обычно прослеживается некий «главный» стиль — высокоравневая структура системы, которая и связывает компоненты, потенциально написанные очень по-разному.

Архитектурные стили используются, чтобы получить следующие преимущества.

- Переиспользование архитектуры. Создание архитектуры может быть сложной задачей, особенно когда у вас есть только требования и «чистый лист». Выбор стиля сужает пространство решений и направляет архитектурную мысль. При этом для новых задач можно применять хорошо известные и изученные решения, обладающие известными достоинствами и недостатками применительно к вашей ситуации.
- Переиспользование кода. Часто у архитектурных стилей бывают неизменяемые части, которые можно один раз реализовать, а затем переиспользовать в каждой системе. Трёхзвенная архитектура, например, реализуется многими библиотеками для разработки веб-приложений, для событийно-ориентированных стилей есть хороший middleware (например, уже упоминавшийся ROS), для распределённых стилей — технологии наподобие gRPC или WCF, и т.д. Это существенно сокращает затраты на разработку.
- Упрощение общения и понимания системы. Как и в случае с паттернами, достаточно просто назвать стиль по имени и не надо объяснять, как что устроено — опытные разработчики вас сразу поймут.
- Упрощение интеграции приложений, на тактическом уровне за счёт переиспользования стандартов и middleware, типичных для стиля, на стратегическом — за счёт того, что понятно, куда и как встраиваться, не нарушив архитектурные ограничения каждого из приложений.
- Применение специфичных для стиля методов анализа. Поскольку стили накладывают ограничения на структуру систем, иногда эти ограничения достаточно жёсткие, чтобы про систему можно было что-то доказать или что-то посчитать. Хороший пример в этом плане — стиль Pipes and Filters, к которому применимы алгоритмы анализа графов, которые можно использовать для анализа пропускной способности системы, её надёжности и т.п.
- Специфичные для стиля методы визуализации — тот же Pipes and Filters удобно рисовать с помощью визуальных языков, ориентированных на данные, таких



© N. Medvidovic

как Data Flow Diagram. Или становится возможным использование предметно-ориентированных языков — например, на том же Pipes and Filters построено программирование в LabVIEW и Matlab/Simulink. Pipes and Filters тут не исключение, например, Microsoft Robotics Developer Studio имела по сути визуальный DSL для создания программ в распределённом веб-сервисном стиле.

Все стили фиксируют ограничения на возможные архитектуры, и какие именно эти ограничения — и есть описание стиля. Стили определяются тремя основными соображениями: набор используемых в стиле элементов, набор правил, по которым эти элементы соединяются, и семантика, стоящая за элементами. Элементы могут быть компонентами, соединителями, элементами данных и т.п., например, некоторые стили описывают объекты и вызовы методов, некоторые — сервера и каналы связи, некоторые — фильтры и каналы данных. Правила соединения элементов — это набор «топологических» ограничений на то, кто с кем может соединяться, и это, как правило, и есть самая суть стиля. Например, строгий слоистый стиль требует, чтобы элементы одного слоя могли общаться только друг с другом и с элементами слоя ниже. Семантика, стоящая за элементами, ограничивает их возможности, например, фильтрам можно запретить иметь собственное состояние, а каналам — преобразовывать данные.

Дальнейшее рассмотрение будет вестись на одном примере (далеко не для всех стилей подходящем, но так даже веселее) — игре «посадка на луну», распространённой на игровых автоматах в 80-е.

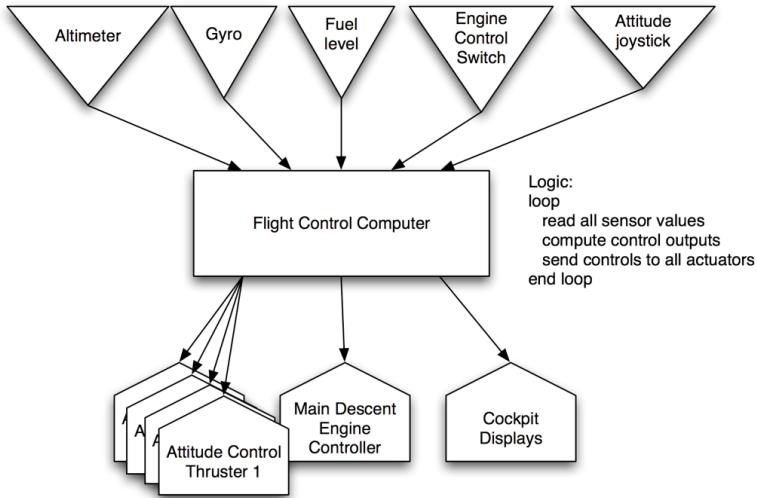
Суть игры в том, что у вас есть спускаемый аппарат с двигателем, тягой которого можно управлять, ограниченный запас топлива и необходимость мягко (с вертикальной скоростью не больше заданной) посадить аппарат на поверхность планеты. При этом при старте игры задано количество топлива и начальная скорость аппарата — естественно, так, чтобы топлива хватало как раз на оптимальную посадку, поэтому игра на самом деле может быть весьма сложной.



© N. Medvidovic

Продвинутые версии позволяли также управлять горизонтальным движением аппарата (например, поворотом маршевого двигателя), что делало игру ещё сложнее, потому что надо было выбрать горизонтальную поверхность для посадки.

Например, Lunar Lander в Sense-Compute-Control-стиле мог бы иметь такую архитектуру:



© N. Medvidovic

Есть компоненты, отвечающие за симуляцию сенсоров аппарата, есть борткомпьютер, отвечающий за вычисление управляющего воздействия, есть актуаторы — маневровые двигатели, маршевый двигатель, дисплеи приборов в кабине. Игра на игровых автоматах такую глубину симуляции, наверное, не обеспечивала, но если бы мы хотели сделать максимально реалистичную игру, её компонент, отвечающий за симуляцию самого аппарата, мог бы выглядеть как-то так.

Дальше речь пойдёт про следующие стили.

- “Традиционные”, связанные с языком:
  - главная программа/подпрограммы — наследие структурных языков программирования, до сих пор вполне применяется при написании небольших утилит, драйверов и т.п.
  - объектно-ориентированный — точнее «сырой» объектно-ориентированный стиль, предполагающий докомпозицию программы на взаимодействующие объекты без дополнительных ограничений. Многие другие стили уточняют этот.
- Уровневый стиль и его уточнения:
  - виртуальные машины,
  - уже знакомая нам трёхзвенная архитектура,
  - клиент-сервер.
- Стили, ориентированные на поток данных:
  - пакетное исполнение,
  - каналы и фильтры.

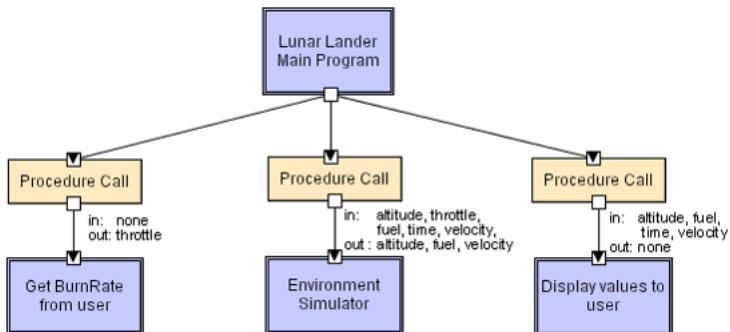
- Peer-to-peer — стиль разработки распределённых приложений. Сегодня про него упомянем, как дойдём до распределённых приложений, рассмотрим подробнее.
- Стили с общей памятью, более конкретно — Blackboard.
- «Интерпретатор» и «мобильный код» — стили, строящиеся вокруг исполнения стороннего кода в наиболее подходящем для этого месте.
- Стили с неявным вызовом:
  - «чистый» событийно-ориентированный стиль,
  - событийно-ориентированный стиль с шиной событий,
  - «Издатель-подписчик».
- «Производные» стили, являющиеся уточнением и комбинацией существующих:
  - распределённый объектно-ориентированный стиль (идея за технологиями типа CORBA),
  - REST,
  - Components and Connectors (C2).

### 3.1. Главная программа/подпрограммы

Главная программа/подпрограммы — стиль, опирающийся на функциональную декомпозицию. Задача разбивается на подзадачи, каждая из которых решается отдельной функцией, которая в свою очередь может вызывать другие функции и т.д., пока задачи не станут достаточно простыми. Стиль хорош простотой и предсказуемостью поведения системы, скоростью работы. Плох тем, что для любой нетривиальной задачи функций (и даже уровней декомпозиции) получится существенно больше, чем можно удержать в голове, и между ними потребуется передавать данные (опять-таки, в очень больших количествах), которым очень сложно обеспечить инкапсуляцию. В общем, такой подход реально хорошо подходит для небольших систем, но если получается больше нескольких тысяч строк кода, всё становится плохо управляемым.

Формально компонентами в таком стиле являются функции, соединителями — вызовы, с передачей параметров, ограничениями — обычно граф вызовов должен образовывать ориентированный ациклический граф, но при необходимости колбэков и взаимной рекурсии это ограничение может нарушаться (что несколько усложняет анализ программы).

Lunar Lander в таком стиле мог бы быть спроектирован так:



© N. Medvidovic

### 3.2. Объектно-ориентированный стиль

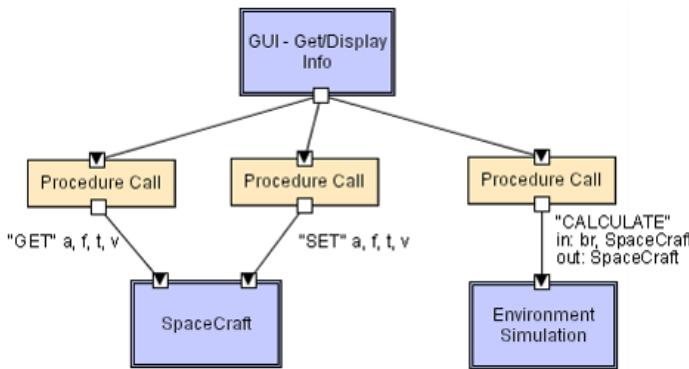
«Сырой» объектно-ориентированный стиль — когда программа представляется в виде набора взаимодействующих объектов в обычном объектно-ориентированном стиле. Компонентами в этом стиле выступают объекты, соединителями — вызовы методов, ограничения — объекты должны полностью контролировать своё внутреннее состояние и менять его можно только через вызовы методов, при этом реализация этих методов должна быть полностью скрыта от других объектов. В общем, обычные требования ООП.

Я бы вообще не называл объектно-ориентированный стиль архитектурным стилем, поскольку он почти никаких ограничений не накладывает, но тем не менее, можно говорить об его достоинствах и недостатках. Достоинства — прежде всего, близость к предметной области и «обычному» человеческому мышлению. Сущности предметной области и так имеют своё состояние и поведение, это более-менее прямолинейно ложится в код. Кроме того, с точки зрения техники программирования каждый объект независим, так что его можно разрабатывать отдельно, и главное, думать о нём отдельно: рассматривать систему как набор взаимодействующих агентов, которые вместе решают задачу, но каждый из них более-менее обособлен, его можно отдельно разрабатывать и отдельно тестировать. Ещё одно важное тактическое соображение — реализацию объектов можно смело менять, пока выполняются их инварианты, во внешнем мире это гарантированно ничего не сломает.

Есть и недостатки. Главный — это, пожалуй, отсутствие строгих топологических ограничений на связи между объектами, что приводит к тенденции «чистых» объектно-ориентированных программ превращаться в месиво из объектов, где каждый вынужден знать о каждом. Поэтому при программировании в чистом объектно-ориентированном стиле обязательно создание высокоуровневой структуры системы или применение дополнительных стилей, которые её диктуют, например, слоистого.

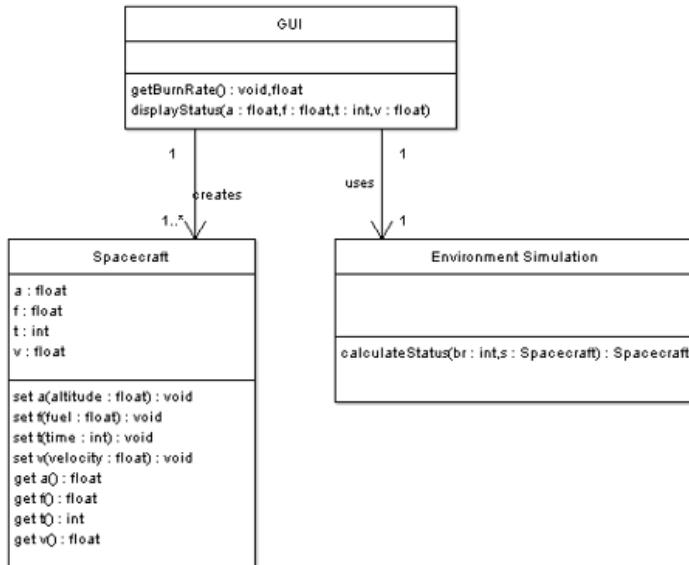
Второй недостаток (по сравнению с функциональным программированием, по крайней мере, и даже со структурным) — склонность к побочным эффектам. Методы обычно меняют состояние объекта, так что разная последовательность вызовов одних и тех же методов с одинаковыми параметрами приводит к разным результатам. Это существенно усложняет анализ.

Lunar Lander в таком стиле мог бы быть спроектирован так:



© N. Medvidovic

Или, в более формальной нотации диаграмм классов UML:



© N. Medvidovic

### 3.3. Слоистый стиль

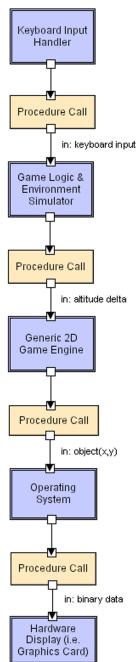
Перейдём к рассмотрению «настоящих» архитектурных стилей. Первый такой стиль, а точнее даже группа стилей — это слоистый стиль и его возможные вариации. Суть этого стиля в том, что мы разделяем систему на слои, где каждый слой может пользоваться слоями ниже и предоставляет интерфейс для слоёв выше, при этом сам ничего о них не зная. Его можно понимать как «многоуровневый клиент-сервер» в том смысле, что каждый слой выступает клиентом слоёв ниже и сервером для слоёв выше. Компонентами в таком стиле выступают сами слои, они могут быть сколь угодно сложно устроены внутри, но это их детали реализации. Соединителями — протоколы общения слоёв (или просто программные интерфейсы). Примеры слоистых архитектуры мы уже видели: трёхзвенная архитектура, также по такому принципу устроены сетевые стеки (модели OSI и TCP/IP состоят из слоёв протоколов), операционные системы, многие бизнес-приложения. Lunar Lander в таком стиле мог бы выглядеть так, как на рисунке справа.

Преимущества слоистого стиля — это в первую очередь постепенное повышение уровня абстракции от низких уровней к высоким. В строгом варианте, когда слой может общаться только со слоем непосредственно ниже, это позволяет вообще не думать о реализации всей системы, а просто программировать в терминах предоставляемой слоем абстракции (так, например, реализуются сетевые приложения, вы просто пользуетесь сокетами или высокоуровневыми протоколами и не интересуетесь тем, что происходит на уровнях глубже).

Ещё слоистость существенно облегчает сопровождение системы. Поскольку каждый уровень влияет только на уровня выше, влияние каждого изменения легко оценить и отследить. При этом можно использовать разные реализации каждого уровня, лишь бы они удовлетворяли общему интерфейсу — примером этого снова являются сетевые приложения, мы можем использовать на физическом уровне Ethernet или WiFi, совершенно ничего не меняя уровнями выше. То же касается драйверов операционной системы — мы можем использовать низкоуровневую графическую библиотеку типа OpenGL для вывода графики, не зная, что за видеокарта у пользователя, был бы драйвер. Да и саму библиотеку можно поменять (на DirectX или Vulcan), если у нас есть высокоуровневый графический движок, обрабатывающий низкоуровневые интерфейсы (например, Unity или UnrealEngine).

Однако есть и проблемы. Во-первых, уровневый стиль оказывается не всегда применим — взаимодействие между элементами системы может быть таким, что не позволяет себя упорядочить по уровням. Пример тому был на первой лекции, с ПО к осциллографу, там уровневый стиль формально можно было навести, но данные шли «вверх» по уровням, а команды от пользователя — «вниз», так что все части системы всё равно были вынуждены знать про все остальные. Такие ситуации в реальной жизни встречаются, и тогда пытаются натянуть уровневый стиль на систему не стоит.

Во-вторых, проблемой может стать производительность системы. Сложные уровневые архитектуры имеют тенденцию обрастиать функциями, которые просто прокидывают запрос на уровень ниже, что ведёт к ненужному оверхеду на вызовы. Если производительность критична, уровневый стиль может всё-таки хорошо подойти, но может и нет (осо-



© N. Medvidovic

бенно, если перестараться с количеством уровней).

Тем не менее, уровневость — это хорошо, поэтому иногда стоит даже проделать дополнительную работу, чтобы разделить систему на уровни. Уровневая архитектура считается довольно стандартной и, например, профстандарт профессии «архитектор» даже подразумевает её как архитектуру по умолчанию.

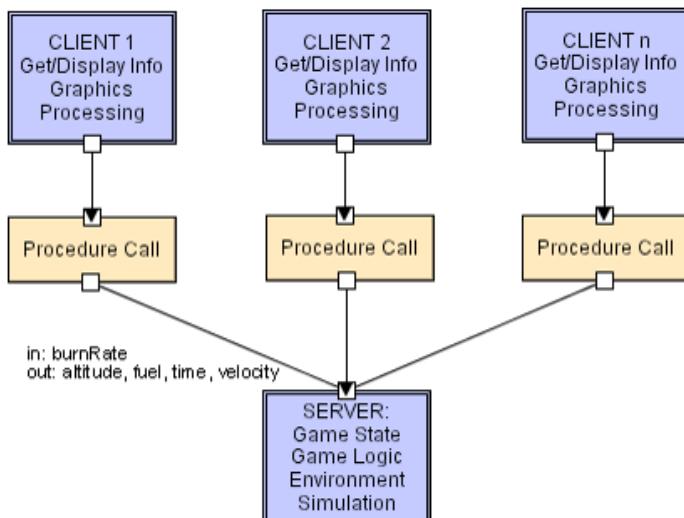
## 3.4. Клиент-сервер

«Клиент-сервер» — это в каком-то смысле вырожденный случай уровневой архитектуры, когда уровня всего два. Собственно, клиенты и сервер — это компоненты такого архитектурного стиля, сетевые протоколы (обычно) — соединители. Ограничения — клиенты не могут общаться друг с другом и могут общаться только с сервером, сервер ничего не знает о клиентах до того момента, как они не начнут с ним взаимодействовать, даже их количество.

Серверов, кстати, может быть много, это позитивно сказывается на масштабируемости системы, но привносит дополнительные трудности синхронизации состояния серверов, о которых будет чуть подробнее позже, в части курса, относящейся к распределённым приложениям.

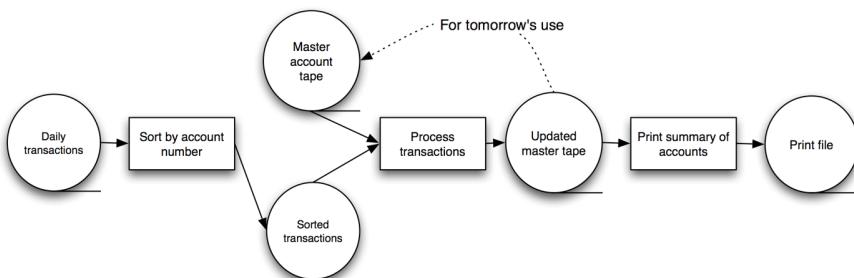
Стиль типичен для несложных веб-приложений, где клиентом обычно выступает браузерная часть приложения, мобильных сетевых приложений или, что может быть несколько неожиданно, операционных систем. Графическая подсистема Linux, например, реализована по клиент-серверной архитектуре, есть оконный сервер и приложения, которые шлют ему запросы на отрисовку графических примитивов.

С помощью такого стиля можно сделать мультиплейерный Lunar Lander:



### 3.5. Пакетная обработка

Пакетная обработка — «прадедушка архитектурных стилей», применявшейся ещё на заре массовой информатизации в финансовых системах глубокой древности:



© N. Medvidovic

Данные в те времена хранились на магнитных лентах (а это медленные устройства с последовательным доступом, перемотать ленту было целым делом). Данные о транзакциях за день подавались на вход программе сортировки, которая сортировала их по номерам аккаунтов, после чего выгружались на ленту с отсортированными транзакциями. Затем они и лента с данными об аккаунтах подавались на вход программе, которая проводит транзакции и сливает их в обновлённую ленту с аккаунтами (как в сортировке слиянием, для этого и надо было сначала транзакции отсортировать). Затем то, что получилось, подавалось на вход программе-печаталке, которая выдавала статистику по аккаунтам.

Итого, система в стиле «пакетная обработка» строится как набор отдельных программ, которые выполняются последовательно, обмениваясь данными средствами операционной системы. Это давно уже не ленты, а файлы, либо pipes или named pipes<sup>2</sup>. При такой схеме между процессами требуется передавать в явном виде всё, что необходимо им для работы — сами данные, конфигурацию, управляющие команды.

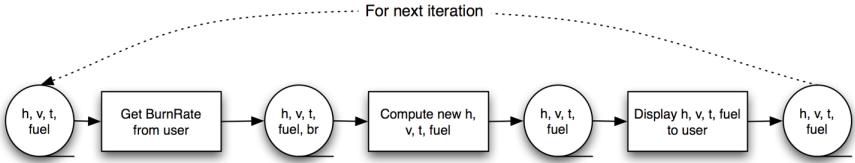
Стиль хоть и древний, но очень популярен до сих пор. По сути, Linux way с большим количеством маленьких утилит, из которых с помощью пайпов выстраиваются конвейеры — это и есть пакетная обработка. Так что этот стиль применяется практически во всех скриптах под Linux и без него трудно представить работу системных администраторов, DevOps и т.д.

Важное преимущество этого стиля — независимость отдельных программ. Они могут быть написаны на каком угодно языке и какой угодно технологии, лишь бы умели читать из входного потока. Можно их вообще не писать, а переиспользовать уже готовые в любой части конвейера. Этакая микросервисная архитектура внутри одной машины.

Важный недостаток этого стиля — неторопливость работы. Каждая программа — отдельный процесс, даже просто их запуск трудоёмок по времени и памяти, да и коммуникации между процессами, хоть и не так тяжелы, как коммуникации по сети, всё-таки гораздо медленнее общения потоков внутри процессса.

Lunar Lander в таком стиле мог бы выглядеть вот таким образом:

<sup>2</sup> Абстракция чего-то среднего между файлом и очередью сообщений, в ней можно писать одним процессом и вычитывать данные другим. Реально данные на диск не сохраняются, поэтому пайпы гораздо быстрее файлов как средство общения между процессами. Кстати, пайпы (даже именованные) поддерживаются и Windows, просто в Linux ими пользоваться удобнее



© N. Medvidovic

Не уверен, что такая архитектура подошла бы для интерактивного приложения, но, ээ, как Play By Email бы, наверное, было неплохо. Некоторые пошаговые стратегии прошлого очень хорошо поддерживали Play By Email, хотя и вряд ли использовали пакетный стиль как архитектуру.

### 3.6. Каналы и фильтры

Каналы и фильтры (или «pipes and filters») — стиль, в котором программа представляется в виде набора фильтров, которые как-то преобразуют данные, идущие по каналам. При этом фильтры независимы друг от друга, то есть не имеют разделяемого состояния и ничего не знают про фильтры до и после них. Всё, что они видят — это данные в своих входных каналах. Собственно, фильтры являются единственным типом элементов в такой архитектуре, а каналы — единственным типом соединителей.

«Каналы и фильтры» похожи на «пакетную обработку», но не требуют, чтобы каждый фильтр был отдельной программой, что помогает победить проблемы с производительностью в пакетной обработке. Кроме того, каналы и фильтры имеют тенденцию образовывать сложные сети, в отличие от пакетной обработки, где всё в основном линейно.

Бывают варианты каналов и фильтров:

- конвейеры — где фильтры связаны просто в линейную цепочку, очень топологически простой стиль, подходящий для несложной логики обработки (хотя сами фильтры могут быть сколь угодно сложны);
- ограниченные каналы — где канал представляет собой очередь с ограниченным количеством элементов, блокирующую фильтр-источник, если очередь переполнена. На самом деле, лучше ограниченность каналов иметь в виду всегда, потому что фильтры, обрабатывающие данные с разной скоростью, могут привести к «пробкам» из данных на разных этапах обработки.
- Типизированные каналы — где каналы знают тип передаваемых данных, и фильтры могут подключаться только к каналам правильного типа. Именно такой стиль в итоге был выбран в первой лекции этого курса в примере про осциллограф.

Преимущества этого стиля таковы.

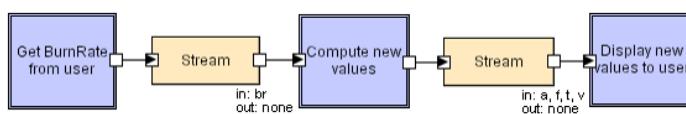
- Поведение системы — это просто последовательное применение поведений компонентов. Так что о нём легко рассуждать, его легко понять, такие системы легко поддерживать.

- Легко добавлять, заменять и переиспользовать фильтры. Если не принимать в расчёт типизированные каналы, то вообще любые два фильтра можно использовать вместе. Если принимать, то любые два фильтра, у которых подходящие типы «портов», можно использовать. Специально продумывать интеграцию компонентов не нужно, она получается сама собой.
- Широкие возможности для анализа. Поскольку есть чёткие ограничения на потоки данных, систему можно рассматривать просто как граф из фильтров с рёбрами-каналами, что делает применимыми все алгоритмы анализа графов. Можно считать пропускную способность системы, задержки (среднюю и максимальную), искать взаимные блокировки в сложных сетях.
- Широкие возможности для параллелизма. Каждый фильтр может работать одновременно со всеми остальными, либо в отдельном потоке, либо в отдельном процессе на другой машине (что, кстати, делает фильтры естественными кандидатами в микросервисы).

Недостатки тоже есть:

- Последовательное выполнение — что странно противоречит достоинству про параллелизм. Но пока первые фильтры из сети не сделают своё дело, следующие за ними о работе приступить не могут. Это не важно, когда данных много и вся сеть занята их обработкой, но если данные поступают лишь иногда, они должны последовательно пройти через все фильтры, при этом большая часть фильтров будет простаивать.
- Проблемы с интерактивными приложениями, поскольку данные идут по фильтрам в одном направлении и непонятно, как ими управлять. Можно придумать «обратные» каналы управления, как это было в примере из первой лекции, но об этом надо специально думать и это несколько портит стройную картину этого стиля.
- Пропускная способность всей системы определяется самым «узким» элементом. Опять-таки, это можно обойти, масштабировав медленный фильтр, но это может быть технически непросто и об этом надо вовремя подумать.

Lunar Lander в этом стиле мог бы быть спроектирован вот так:



© N. Medvidovic

## 3.7. Blackboard

Blackboard — это архитектурный стиль с общей памятью. Есть центральное хранилище данных, тот самый «Blackboard», есть компоненты, которые знают только про Blackboard, не имеют своего состояния и, собственно, выполняют полезную работу. Процесс работы системы устроен так, что каждый компонент смотрит на Blackboard, (возможно) находит там данные, которые может преобразовать, преобразовывает их и записывает обратно на

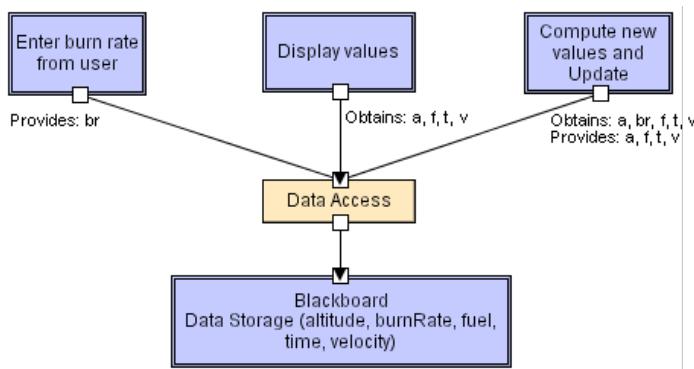
Blackboard, в надежде, что какой-то другой компонент сможет преобразовать их дальше. Это чем-то похоже на группу экспертов, которые решают задачу, сидя в одной комнате с доской — каждый эксперт шарит только в своей предметной области, поэтому выходит к доске и пишет на ней что-то, что соответствует его экспертизе и, возможно, поможет другим экспертам решить задачу.

Весь процесс вычислений управляет только через Blackboard, поэтому если мы хотим, чтобы некоторые действия выполнялись последовательно, нам надо хранить на Blackboard какой-то токен исполнения и обновлять его при выполнении действий.

Такой подход хорош тем, что компоненты могут быть полностью независимы и работать параллельно (и разрабатываться независимо, кстати), система легко масштабируется и расширяется добавлением новых компонентов, и главное, что вам не надо даже понимать алгоритм решения задачи — вы просто бросаете в систему компоненты пока задача не решится. Это же и главный недостаток этого стиля — задача легко может и не решиться. Поэтому такой стиль применяется прежде всего в системах искусственного интеллекта, где задача и так вполне может быть нерешаемой. Ещё, кстати, Blackboard, как правило, является узким местом системы, поскольку ему надо синхронизировать доступ независимым компонентам к себе.

Кстати, родственен Blackboard стиль, основанный на правилах переписывания — например, машины Маркова и язык Рефал, графовые грамматики и подобные штуки. Там тоже есть центральная структура данных и набор правил, которые её независимо и постепенно изменяют. Такой подход вполне распространён и в «обычной» программной инженерии.

Lunar Lander в стиле Blackboard мог бы быть спроектирован вот так:



© N. Medvidovic

### 3.8. Стили с неявным вызовом

Стили с неявным вызовом — общее название разных вариантов событийно-ориентированных стилей или вообще стилей, где хотя бы иногда используются оповещения вместо явных вызовов методов. Во всех таких стилях есть «слушатели», которые могут подписываться на события, и при наступлении события система сама вызывает всех зарегистрированных слушателей.

В таких архитектурах компоненты имеют два вида интерфейсов — обычный набор методов, и события, на которые можно подписываться. В качестве соединителей использу-

зуются либо прямые вызовы методов, либо неявные вызовы слушателей по наступлению события (почему стили и называются стилями с неявным вызовом).

Инварианты всех таких стилей:

- те, кто производит события, не знают, кто и как на них отреагирует — они, как правило, технически имеют список подписавшихся, но обычно не вправе даже узнать их количество, не говоря уж о том, чтобы что-то делать с подписавшимися напрямую;
- не делается никаких предположений о том, как событие будет обработано и будет ли вообще — источник просто нотифицирует систему о наступлении события, а уж подписан на него кто-нибудь или нет, в каком порядке кто подписан и т.д. — не его дело.

Преимущества всех таких стилей — это переиспользуемость компонентов и лёгкость конфигурирования системы. Высокая переиспользуемость достигается за счёт очень низкой связности между компонентами, ведь источник событий вправе вообще ничего не знать о тех, кто им пользуется. Лёгкость конфигурирования, как во время компиляции, так и во время выполнения, достигается за счёт того, что подписки на события можно легко менять, меняя при этом всю функциональность системы.

Недостатков, тем не менее, тоже довольно много.

- Зачастую неинтуитивная структура системы. Без применения дополнительных архитектурных ограничений подписки на события превращаются в хаотичный клубок, в котором хаотично распространяются нотификации. Поэтому мы и рассмотрим конкретные стили, накладывающие дополнительные ограничения.
- Компоненты не управляют последовательностью вычислений. Работа системы состоит в генерации событий и реакций на события, и делать что-то в правильном порядке в сколько-нибудь сложной системе может оказаться проблематичным.
- Непонятно, кто отреагирует на запрос и в каком порядке придут ответы. Компонент, генерирующий события, не вправе предполагать, что на событие кто-то отреагирует, поэтому если это событие, например, «мне нужны данные для дальнейшей работы», мы не вправе рассчитывать на ответ. А если надо запросить несколько разных источников, то неизвестно, кто и когда ответит. Поэтому такие системы принципиально асинхронны.
- Тяжело отлаживаться. Вы не можете просто сделать `step into` при вызове метода, вы должны мучительно ковыряться в списке подписчиков. Некоторые среды, типа C#/Visual Studio, хорошо поддерживают отладку событий, но даже там, когда управление прыгает по всему коду, отлаживаться тяжело. К тому же, событийные системы принципиально асинхронны, что создаёт дополнительную боль.
- Ситуации, очень похожие на гонки, даже если у вас всего один поток. Классическая гонка — это когда результат работы программы зависит от случайного порядка переключения потоков планировщиком. Гонка в событийных системах — это когда результат работы программы зависит от случайного порядка вызова обработчиков при нотификации. Обычно событийные системы хоть и не позволяют закладывать на определённый порядок вызова обработчиков, всё же вызывают их в порядке

подписывания, что делает процесс хоть сколько-нибудь детерминированным. Но, во-первых, это не всегда возможно (например, в распределённых системах — кто первый получил событие, тот его и обработал), во-вторых, порядок подписывания тоже такой себе ориентир — на событие могут подписываться разные компоненты в разных частях кода, и помнить, кто когда должен подписаться, может оказаться слишком хлопотно. В таких местах особо часто появляется и особо опасен анти-паттерн «Sequential coupling».

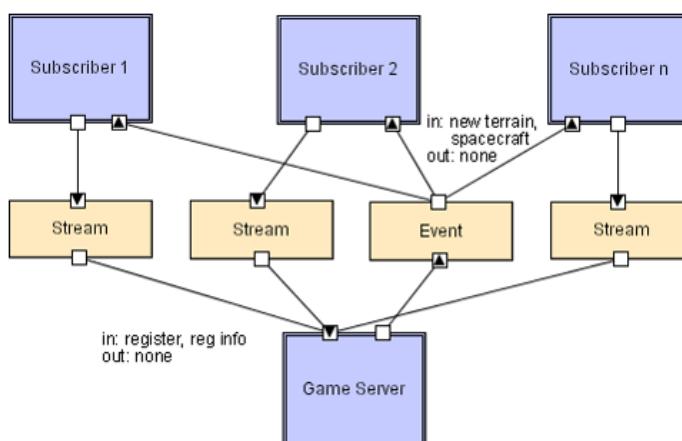
### 3.8.1. Издатель-подписчик

«Издатель-подписчик» — самый простой стиль из стилей с неявным вызовом, и вместе с тем весьма популярный и эффективный. Есть издатели, они публикуют сообщения (синхронно или асинхронно, то есть дожидаясь их обработки либо нет). Есть подписчики, которые подписываются на издателей и получают от них события. Есть маршрутизаторы, задача которых — быть посредниками между издателями и подписчиками, фильтровать и маршрутизировать сообщения. При этом в самых простых конфигурациях без маршрутизаторов прекрасно обходятся, но они могут быть полезны: например, маршрутизатор может по очереди отправлять сообщения то одному подписчику, то другому, реализуя тем самым балансировку нагрузки.

Компонентами в таком архитектурном стиле являются издатели, подписчики и маршрутизаторы, соединителями — сетевые протоколы или *очереди сообщений*, либо, если дело происходит на одной машине, механизмы наподобие паттерна «Наблюдатель». В качестве данных в этом стиле выступают подписки, нотификации о произошедших событиях, публикуемая издателями информация. Ограничения — издатели ничего не знают о подписчиках, подписчики, как правило, ничего не знают друг о друге, только об издателях.

Преимущества этого стиля, как и обычно для событийно-ориентированных схем — очень низкая связность между компонентами, но при этом высокая эффективность распространения информации, отчасти за счёт свойственной этому стилю простоты топологии, отчасти за счёт наличия маршрутизаторов.

Lunar Lander в этом стиле мог бы быть спроектирован вот так:



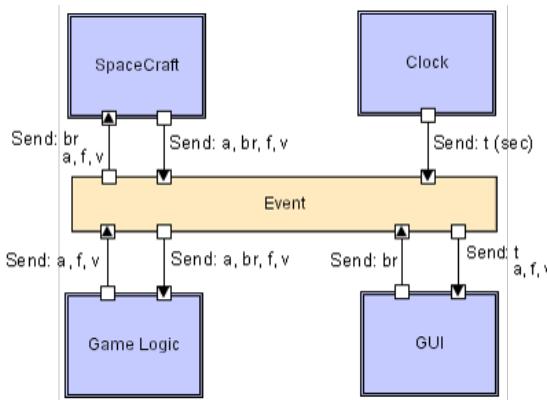
### 3.8.2. Событийно-ориентированный стиль с шиной

Событийно-ориентированный стиль с шиной предполагает наличие шин событий, через которые могут общаться компоненты. При этом всё остальное как в чистых событийно-ориентированных стилях, есть компоненты, которые могут генерировать какие-то события и подписываться на чужие, но разница в том, что компоненты принципиально не могут взаимодействовать друг с другом напрямую, а общаются только через шину. Соответственно, подписаться можно только на события в шине, и посыпать события можно только в шине. При этом возможны два варианта взаимодействия — «push», когда шина сама активно уведомляет своих подписчиков, и «pull», когда компоненты время от времени опрашивают шину на предмет наличия в ней новых событий. Шина в системе вполне может быть одна, но часто это несколько именованных шин с разными типами данных.

Преимущества такого подхода — это ещё большая независимость компонентов, лёгкость масштабирования и добавления новой функциональности. Если кто-то не успевает обрабатывать запросы — не проблема, подключим к шине ещё одну копию. Если нам надо новую функциональность, мы просто пишем компонент, подключаем его к шине и ничего больше в системе обычно менять не надо. Особенно эффективен такой стиль для распределённых приложений, где каждый компонент может быть отдельным сервисом.

Пример дальнейшего уточнения такого стиля — архитектурный паттерн интеграции «Enterprise Service Bus», умная шина, позволяющая преобразовывать данные в универсальное внутреннее представление и выгружать их в виде, пригодном для каждого компонента. Там обычно компонентами выступают третьясторонние приложения, типа электронных таблиц, бухгалтерских систем и т.п. — мы даже не можем менять код компонентов, но с помощью шины можем добиться их эффективной интеграции.

Lunar Lander в таком стиле мог бы быть спроектирован вот так:



© N. Medvidovic

## 3.9. Peer-to-peer

Стиль «Peer-to-peer», или одноранговая сеть, — стиль, при котором система состоит из большого количества приложений, каждое из которых в принципе может само решать все задачи и является самодостаточным, но чем больше приложений работают совместно, тем шире возможности системы. Компоненты в таком стиле — это отдельные приложения,

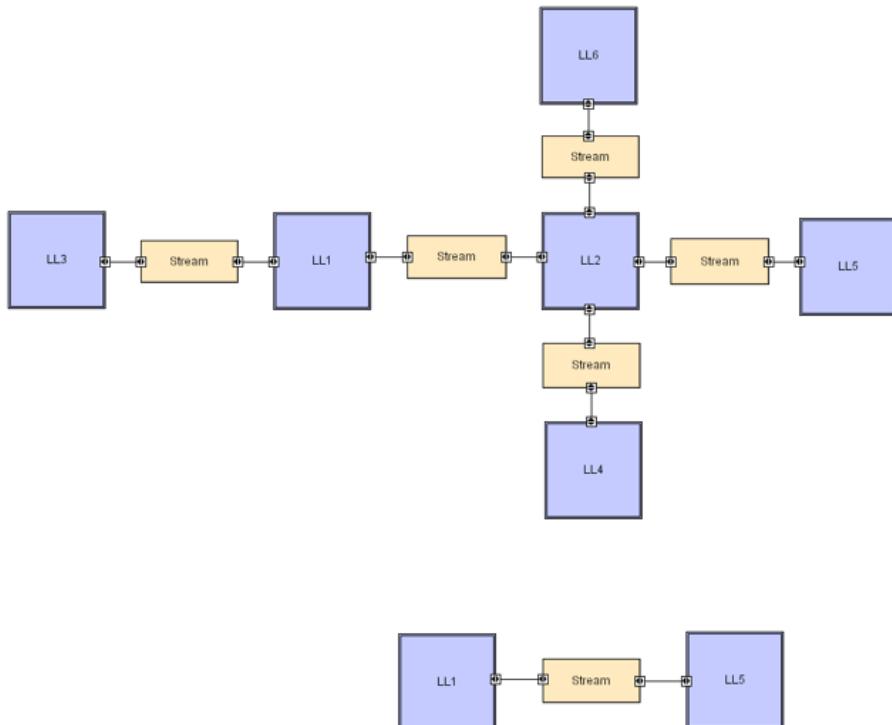
которые имеют своё состояние, свой поток управления и всё, что нужно для работы. В качестве соединителей, как правило, выступают сетевые протоколы, а в качестве элементов данных — сообщения по сети.

Топологических ограничений на связи между компонентами не накладывается, даже наоборот, приветствуются избыточные связи. Кроме того, предполагается, что топология сети может динамически изменяться во время работы. Компоненты могут выходить из сети или появляться новые.

Основное преимущество Peer-to-peer — это отказоустойчивость. Можно хоть физически уничтожить часть системы, остальная часть продолжит работать, хоть и менее эффективно. Ещё такую систему легко масштабировать, просто динамически подключая к сети новые компоненты, что делает peer-to-peer идеальным для распределённых вычислений, особенно в ситуациях, когда каждый участник никому ничего не должен и работает в сети только тогда, когда у него есть возможность.

Самый, пожалуй, известный пример peer-to-peer-системы — это BitTorrent, где каждый узел может работать как файловый сервер, но чем больше узлов, тем больше информации можно хранить и тем быстрее её можно скачивать. Есть и менее известные применения, например, сенсорные сети или стаи беспилотников, применяющиеся в военной или природоохранной сферах. Если один из беспилотников вышел из строя, стая переконфигурируется и продолжает выполнение задания.

Lunar Lander в виде сети из зондов, приземляющихся на Луне, мог бы быть устроен так:



Подробнее про Peer-to-peer мы поговорим, когда будем рассматривать архитектуру распределённых приложений.

## 3.10. Гетерогенные стили

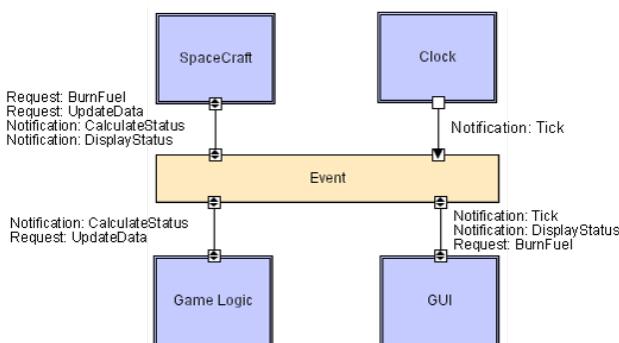
Гетерогенные стили — это условное название для более специфичных архитектурных стилей, являющихся уточнением и соединением нескольких стилей, упоминавшихся ранее. Примеры таких стилей — REST (Representational State Transfer), про который в этом курсе будет позже, Components and Connectors, стиль «распределённые объекты» и его конкретная реализация CORBA. Вообще, CORBA — это конкретный стандарт (к тому же, безнадёжно устаревший на данный момент), но его авторы настолько серьёзно подошли к архитектурным аспектам, что он заслуживает тут упоминания.

### 3.10.1. Components and Connectors (C2)

Components and Connectors — это вариант стиля с неявным вызовом, событийный стиль с шиной плюс уровневый стиль. Есть компоненты — независимые, потенциально параллельные производители или потребители сообщений. Есть соединители — маршрутизаторы сообщений, которые могут фильтровать, преобразовывать и рассыпать сообщения двух видов: нотификации и запросы. Нотификации анонсируют изменения в состоянии, содержат данные про суть изменений. Запросы — запрашивают выполнение действия.

Стиль хорош тем, что накладывает жёсткие топологические ограничения на запросы и нотификации: нотификации шлются только «вниз», а запросы только «вверх». Это и привносит уровневую структуру.

Lunar Lander в таком стиле мог бы быть спроектирован вот так:



© N. Medvidovic

### 3.10.2. CORBA (на самом деле, «распределённые объекты»)

CORBA (Common Object Request Broker Architecture) — это конкретный стандарт, разработанный консорциумом OMG (тем самым, который стоит за UML) аж в начале 90-х. Задачей стандарта было определить протоколы и типовую архитектуру систем, состоящих из объектов, работающих на разных хостах под управлением разных операционных систем и написанных на разных языках программирования. CORBA была реализована в

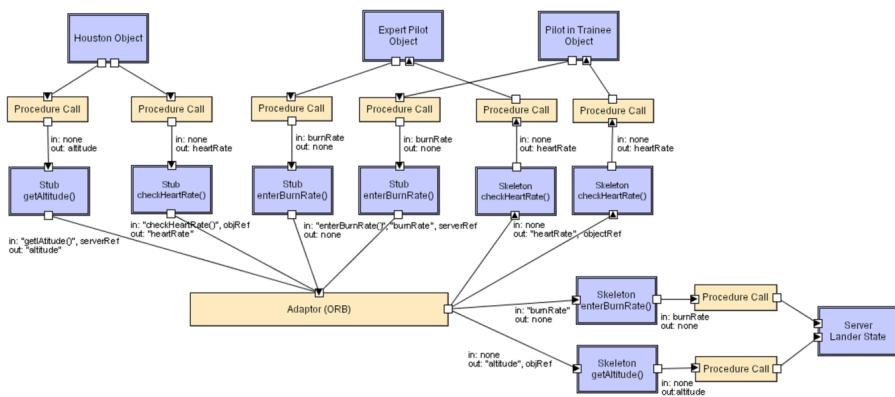
Java и стала очень популярна в информационных системах 90-х, но потом потеснена веб-сервисами на SOAP, а затем и REST-сервисами.

Компонентами в такой архитектуре выступают обычные объекты, каждый из которых запущен, возможно, в своём отдельном процессе и снабжён инфраструктурой для сериализации/десериализации и обработки сетевых запросов (это делает обычно какое-то заранее написанное middleware). В качестве соединителей выступают сетевые протоколы удалённого вызова, связывающие объекты с Object Request Broker (который выступает чем-то вроде шины между объектами, так что CORBA можно считать смесью событийного стиля с шиной и чистого объектно-ориентированного стиля). Никаких других топологических ограничений не накладывается.

При этом есть дополнительные инфраструктурные ограничения, связанные с сетевой природой архитектурного стиля (и делающие идею «а давайте возьмём обычное приложение и сделаем каждый объект веб-сервисом» экстремально плохой, кстати). Во-первых, все данные, передаваемые при вызовах (параметры, возвращаемые значения и исключения) должны быть сериализуемы. Во-вторых, каждый объект должен уметь обрабатывать ошибки сети и спокойно относиться к временной недоступности сотоварышей. То есть каждый удалённый вызов должен сам обрабатывать сетевые ошибки.

Преимущества такого стиля, на самом деле типичные для более-менее любой распределённой архитектуры — это независимость от конкретной технологии реализации. Вы можете каждый объект писать на том языке, который вам больше нравится, лишь бы сервис поддерживал стандарт. Более того, каждый объект может быть физически расположен где угодно, хоть на другом конце света, и работать на самом разном железе. Что даёт масштабируемость, гибкость и т.д., но и добавляет разных проблем типа необходимости аккуратного деплоя.

Lunar Lander в таком стиле мог бы выглядеть как-то так:



© N. Medvidovic

Про распределённые объекты мы тоже подробнее поговорим ближе к концу курса, когда речь пойдёт про архитектуру распределённых приложений.

# Domain-Driven Design

Юрий Литвинов

yurii.litvinov@gmail.com

## 1. Введение

Domain-Driven Design (или предметно-ориентированное проектирование) — популярная методология проектирования ПО, которая основана на анализе предметной области и реализации модели предметной области в приложении. Архитектуру в рамках этой методологии предлагается строить не исходя из сиюминутных потребностей реализации, а вокруг “смыслового ядра”, которое отражает основные сущности реального мира, с которым будет работать программа (или семейство программ).

Модель предметной области выражается прежде всего в коде — сущности предметной области становятся классами языка программирования, используемого для реализации. Также для обсуждения предметной области с экспертами и для документирования модели часто используются диаграммы.

Немаловажную роль как в создании модели, так и в её “фиксации” и улучшении играет ещё и устное общение. Неудачные названия классов или методов, неуклюжие и непонятные описания взаимодействий прекрасно слышны в разговоре и являются поводом для того, чтобы посмотреть на модель ещё раз. Сам естественный язык часто подсказывает правильные архитектурные решения — что неудивительно, язык сотни лет оттачивался как раз для того, для чего он нужен в DDD — для передачи сути понятий. Ещё раз напомню, что архитектура — это больше про понимание программы человеком, а не понимание программы компьютером, так что всякие гуманитарные вещи могут сильно помочь при разработке архитектуры.

Модель определяет единый язык, на котором должны общаться все члены команды и эксперты предметной области, которые им помогают. Если какой-то термин зафиксирован как имя класса, использование его синонимов ни в диалогах, ни в документации, ни тем более в коде не допускаются, можно использовать только имя класса. Так же и с методами — если действию дали название, можно использовать только его, и если это не удобно, название меняют. Это, во-первых, упрощает общение и сопровождение программы, во-вторых, способствует улучшению модели, постоянно её тестируя на предмет неконсистентностей, недопониманий и неоднозначностей.

Модель появляется не только благодаря применению единого языка и последовательного именования всех нужных сущностей, она также “выкристаллизовывается” в процессе непрерывного рефакторинга и уточнения. Поскольку программисты редко владеют предметной областью, в которой они решают задачу, построить правильную и хорошую модель предметной области невозможно просто потому, что знаний не хватает. Есть понятие “переработка знаний” — когда программисты строят модель, одновременно активно изучая

предметную область. Узнав что-то новое, они включают это в модель, рисуют диаграммы, обсуждают их с экспертами, корректируют модель, и т.д. до тех пор, пока не будет достигнуто достаточное понимание. Цель “переработки знаний” — не получить понимание, достаточное для реализации программы, а понять принципы, по которым работает сама предметная область, а уже потом написать программу. Интересно, что если модель хороша и программисты неплохи в выделении абстракций, то в процессе переработки знаний кое-чему научиться могут и эксперты.

Дальнейший рассказ является, по сути, кратким пересказом книги Эрика Эванса, “Предметно-ориентированное проектирование. Структуризация сложных программных систем”. М., “Вильямс”, 2010, 448 стр., по крайней мере, первых трёх её частей. Если есть время, лучше её прочитать, если нет, то по этому конспекту, я надеюсь, можно составить общее представление. Некоторые примеры и все картинки взяты оттуда.

## 2. Единый язык

Единый язык — одно из ключевых понятий предметно-ориентированного проектирования. Необходимость его введения связана с тем, что программисты и специалисты предметной области разговаривают на разных профессиональных жаргонах и изначально не понимают друг друга. Если программисты будут общаться в терминах реализации, экспертам это будет не очень полезно. Причём, что особенно опасно, непонимание будет, скорее всего, скрытым — они будут чувствовать себя, как типичные студенты на лекции по матанализу: вроде общий смысл улавливается, но местами происходит что-то такое, с чем, кажется, можно будет разобраться потом, прочитав конспект. Но лекцию по матанализу ведёт человек, разбирающийся в предмете, а общающийся с заказчиком программу программист, возможно, просто неправ, или подсознательно маскирует своё собственное непонимание техническими терминами, отчего и эксперт его не понимает, но не может поправить.

На самом деле, и среди разработчиков одной группы вырабатывается свой жargon, который может быть непонятен новичкам и даже опытным членам соседних групп, работающих над тем же проектом. Это плохо, во-первых, тем, что необходимость постоянно переводить понятия с одного языка на другой вызывает “размытие” этих понятий и некоторую неопределённость, причём, каждый может делать перевод по-своему, что усиливает хаос. Во-вторых, это плохо тем, что один термин может использоваться в разных (иногда очень слегка разных) значениях — внутри проекта возникают “ереси”, что приводит уже к откровенной путанице и багам в коде.

Поэтому предметно-ориентированное проектирование предписывает выработать единого языка и использование его повсюду в проекте, от составления технического задания до имён методов, переменных и тестов. В единый язык входят термины предметной области — как правило, они становятся именами классов, отношения между понятиями и действия, которые сущности могут выполнять — это имена методов, также в язык попадают имена паттернов, используемых при проектировании системы, даже если их в предметной области нет (например, “репозиторий” или “спецификация”). Ещё единый язык должен содержать понятия, относящиеся к высокому уровню архитектуре системы, которые напрямую невыразимы в коде — уровни, ограничения на взаимодействие (например, понятие “канал” в архитектурном стиле “каналы и фильтры” может явно в коде никак не

выражаться, но в архитектуре играет ключевую роль).

Единый язык не создаётся мгновенно, он эволюционирует вместе с пониманием предметной области, моделью и кодом, который эту модель реализует. Поэтому за изменениями языка, которые возникают естественно в разговорах (а помним, что общаться можно только на едином языке), надо следить и соответственно рефакторить модель или код. Например, если все начали говорить “связь” вместо “ассоциация” (потому что так короче и благозвучнее), надо переименовать соответствующий класс и в коде.

На самом деле единых языков в проекте может быть много, как бы бредово это ни звучало. Если проект состоит из нескольких более-менее обособленных частей, то почему нет, каждая из них может иметь свой язык. Например, система планирования грузоперевозок вполне может использовать библиотеку алгоритмов на графах, в самой системе связь может называться “перевозка”, а в библиотеке — “ребро”. Если есть чёткая граница между сферами действия разных языков (и разных моделей и даже мировоззрений, которые связаны с языком) и определены правила преобразования одного в другое (выражающиеся в коде в виде фасадов и адаптеров), то почему нет, это помогает уменьшить общую сложность системы. Если чёткой границы нет и модели (и языки) начинают проникать друг в друга, возникает путаница, которая может привести к провалу проекта.

Небольшой пример важности единого языка. Положим, нам надо сделать систему планирования грузоперевозок и мы рассматриваем задачу прокладки маршрута перевозки через какой-то пункт таможенного контроля. Мы везём груз из точки А в точку Б через 0 или больше пунктов таможни, и у нас есть планировщик маршрута, которому можно сказать точки А, Б и таможни, чтобы он выдал расписание перевозки. Модель глазами программиста могла бы выглядеть так:

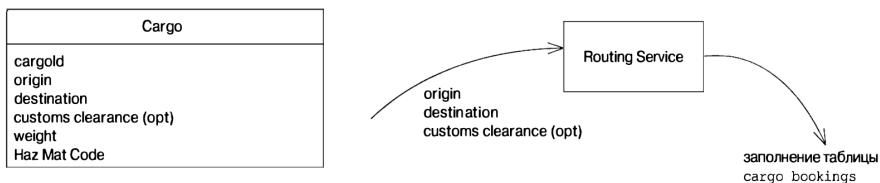


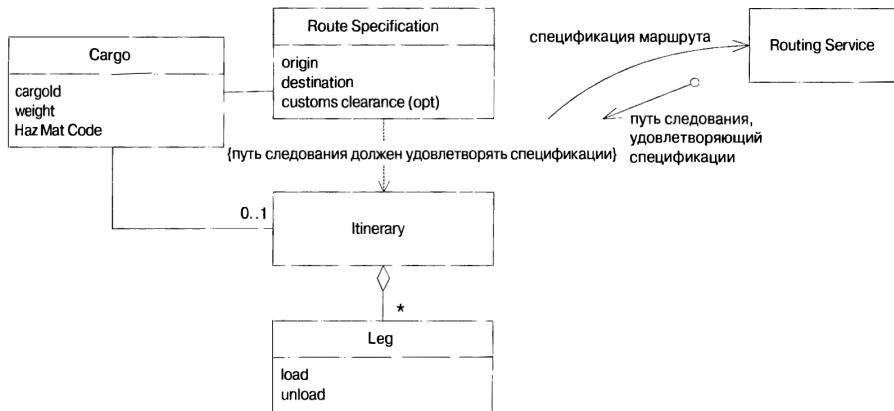
Таблица БД: `cargo_bookings`

Cargo_ID	Transport	Load	Unload

Представьте себе обсуждение с экспертом по логистике вопроса изменения пункта рас��живания:

“Тогда мы удаляем все строки в таблице отправки грузов с заданным идентификатором груза, потом передаем пункт отправки, пункт назначения и новый пункт растаможивания в Маршрутизатор (*Routing Service*) и он заполняет таблицу заново. В объект Груз (*Cargo*) надо добавить логический переключатель, указывающий, есть ли данные в таблице отправки.”

Сравним это с вот такой моделью:



Тут наше обсуждение могло бы выглядеть так:

*“При изменении любого из атрибутов в Спецификации маршрута (Route Specification) мы удаляем старый Маршрут (Route) и просим Маршрутизатор (Routing Service) построить новый маршрут на основе новой Спецификации маршрута.”*

Получилось короче, более точно и даже более полно, потому что в модели выше надо было ещё пояснить, когда сбрасывать маршрут, а тут появилось понятие “Спецификация маршрута”. Вторая модель (и язык, с ней связанный) по сути радикально систему не меняет (**Leg** и **Itinerary** всё также могут быть таблицами в БД), но оставляет гораздо меньше возможностей для взаимного непонимания и ошибок, с ним связанных.

Ещё один хороший пример использования единого языка, из книжки:

*Если передать в Маршрутизатор пункт отправки, пункт назначения, время прибытия, то он найдет нужные остановки в пути следования груза, а потом, ну... запишет их в базу данных.*

*Пункт отправки, пункт назначения и все такое... все это идет в Маршрутизатор, а оттуда получаем Маршрут, в котором записано все, что нужно.*

*Маршрутизатор находит Маршрут, удовлетворяющий Спецификации маршрута.*

Опять-таки, видно, что хорошая модель позволяет строить фразы коротко, точно и без технических подробностей. Но тут ещё видно, как естественный язык помогает процессу разработки архитектуры — мы начали с первой фразы, запнулись, потому что у нас не было сущности “Маршрут”, добавили её в модель, стало лучше, но теперь нам не нравится “и всё такое”, что мы побеждаем добавлением понятия “Спецификация маршрута”. Конечно, натуральный язык и метафоры могут завести не туда, но если это случится, то это скорее проблемы понимания предметной области, и их будет проще найти, чем если бы эти проблемы были бы спрятаны в таблицах в базе данных.

### 3. Модель и реализация

Модель предметной области полезна только в том случае, если она непосредственно связана с кодом. Бывает так, что люди, вообразившие себя крутыми архитекторами, рису-

ют огромную модель предметной области в виде диаграммы классов UML, которую невозможно эффективно реализовать, например, положить в базу данных. Тогда обычно разработчикам приходится частично игнорировать модель, решая свои чисто практические задачи. Это приводит к тому, что код и модель расходятся, при этом модель становится даже вредна, поскольку врёт об устройстве кода.

Бывает и наоборот, люди, вообразившие себя крутыми agile-developerами, начинают “фигачить код” без всякой архитектуры-шмархитектуры. В результате получается программа, полученная хаотичным наслоением слабо связанной функциональности, которую тяжело читать, сопровождать, добавлять новую функциональность. Эванс в книжке кратко описывает два таких проекта и делает интересное наблюдение, что результаты получились примерно одинаковыми.

Из этого следует вывод, что модель всегда должна быть связана с кодом, а добиться этого можно только если каждый архитектор пишет код и любой программист участвует в моделировании. Практика, принятая в некоторых enterprise-проектах, когда аналитики сначала строят модель предметной области, а затем программисты её реализуют, пагубна именно поэтому — модель может игнорировать особенности реализации, а знания аналитиков не будут полностью переданы программистам. И наоборот, новые интересные знания могут быть получены при попытке выразить модель в коде, эти знания аналитикам (или архитекторам) не попадут. В результате программистам придётся проделывать работу по анализу предметной области фактически заново, и “prescriptive architecture” разойдётся с “descriptive architecture”. Есть даже хороший термин “разрушительный рефакторинг”, когда программисты, не понимая глубоко предметную область, рефакторят программу так, чтобы было удобно реализовывать требуемую функциональность. В этот момент, скорее всего, потерянется самая суть программы, её смысловое ядро, и это будет просто работающее приложение, которое будет делать что нужно. DDD же ставит своей целью создавать системы, которые делают больше и лучше, чем нужно (при этом будучи менее трудозатратными при разработке).

Поэтому модели в DDD приходится быть одновременно и моделью анализа, и моделью проектирования. Мы с её помощью пытаемся понять предметную область, и она же используется для того, чтобы писать код (причём, одновременно). Так что если модель окажется полной технических деталей, то анализ предметной области и общение с экспертами неизбежно пострадают, а если она будет слишком абстрактной или непроработанной, то пострадает реализация. Необходим баланс, который достигается обычно за несколько итераций — описываем предметную область, пытаемся её реализовать, получается плохо, правим модель и рефакторим код, пытаемся описать предметную область в новой модели, получается неуклюже, снова рефакторим модель и т.д., пока не придём к модели, которая хорошо служит и коду, и объяснению предметной области.

При этом, разумеется, необходимо, чтобы выбранный язык программирования поддерживал парадигму моделирования — например, с ООП всё довольно легко, а вот с чисто структурными языками, например, С, может быть плохо — в них не выразить естественным образом сущности с состоянием и поведением, свойственные реальному миру. А вот Prolog, напротив, хорош, он близок исчислению предикатов, так что если мы можем построить модель в терминах фактов и правил вывода, на Прологе программа запишется очень естественно.

Ещё не рекомендуется разделять модель, которая служит основой реализации, и модель, которую показывают пользователю. Пример — в Internet Explorer закладки хранились

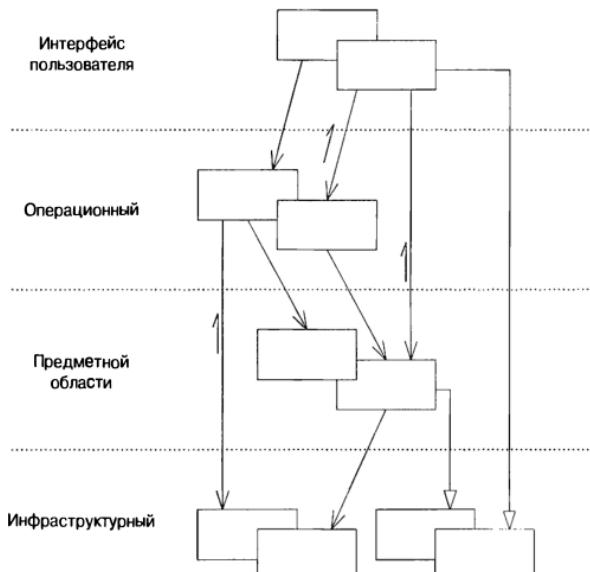
как файлы-ярлыки на диске, но пользователю про это ничего не говорилось. Хочет пользователь сохранить закладку с символом “.”, а ему ошибка — неправильное имя файла. И пользователь не может понять, какого к чёрту файла, он просто закладку сохранил. А вот если бы закладки и показывались как файлы, то пользователь мог бы сортировать их с помощью файлового менеджера или вообще скрипта, и был бы более счастлив.

## 4. Изоляция предметной области

Хорошо, допустим, мы смогли построить модель предметной области. Но этого недостаточно, чтобы получить работающую программу — нужен ещё пользовательский интерфейс, сеть, база данных, всякого рода вспомогательный код, логирование, обработка ошибок, юнит-тесты и т.д. Основная идея DDD в том, что всё это должно быть архитектурно отделено от модели предметной области, чтобы код модели — код, в котором сосредоточена вся суть приложения — мог быть максимально простым, небольшим по размеру и содержащим только существенные для смысла системы вещи. Всё остальное должно находиться в отдельных модулях и просто использовать классы доменной модели.

Самый популярный способ достижения такого разделения (хотя и не единственный) — уровневая архитектура. Программа реализуется в виде набора уровней, где каждый уровень может непосредственно взаимодействовать только с уровнями ниже. Способов разделения на уровни бывает много (трёхзвенная архитектура, семь уровней OSI, четыре уровня TCP/IP и т.д.), DDD требует лишь, чтобы среди всех уровней был уровень предметной области, на котором и сосредоточены все классы модели.

Довольно типичный для информационных систем способ разделения на уровни представлен на картинке:



Уровень интерфейса отвечает только за взаимодействие с пользователем и реализует только отображение и обработку событий. Никакой содержательной логики в нём быть

не должно. Операционный уровень (application layer) занимается координацией действий бизнес-объектов, которые находятся на уровне предметной области. Операционный уровень тоже должен быть очень простым, его ответственность — это инициализировать систему и по сигналу от интерфейса инициировать нужные пользователю операции. Инфраструктурный уровень содержит все вспомогательные вещи, неспецифичные для данной предметной области, например, код работы с базой данных или код работы с сетью.

Уровень предметной области содержит бизнес-объекты, которые ничего интересного кроме, собственно, реализации бизнес-правил, не делают. Их реализация должна быть максимально простой, и именно их реализации следует уделять максимум внимания, потому как именно уровень предметной области определяет конкурентные преимущества и полезность программы.

Операционный уровень специфичен для каждого конкретного приложения, тогда как уровень предметной области может разделяться между несколькими приложениями в семействе. На операционном уровне из классов уровня предметной области собирается то, что делает конкретно наше приложение, операционный уровень же координирует бизнес-объекты. Но бизнес-регламенты (например, последовательность действий в бизнес-процессе) должны быть реализованы на уровне предметной области, потому как они хоть и координируют действия других объектов, но объективно присутствуют в предметной области и могут быть переиспользованы в других приложениях. Операционному уровню также запрещается иметь состояние (кроме, быть может, состояния, необходимого для общения с пользователем приложения, типа прогресса операций).

Инфраструктурный уровень поддерживает все уровни выше. Он может обеспечивать архитектурную среду для всех остальных уровней (например, Java Beans или ROS), может содержать специфичный для приложения код работы с третьесторонними библиотеками и технологиями (например, Object-Relational Mapping). Может показаться противостоящим, что базовые классы для уровней выше могут быть на инфраструктурном уровне (то есть деревья наследования растут вверх), но если подумать, кто о ком больше знает — предок или потомок, то становится понятно, что концептуально всё ок.

Уровням ниже, естественно, иногда требуется общаться с уровнями выше. Даже инфраструктурный уровень, если он не состоит только из библиотек, может инициировать действия на уровнях выше (например, получение сетевого пакета вызывает обработку на операционном уровне). Поскольку общаться напрямую уровни не могут, применяются приёмы “косвенного” вызова — callback-и (или виртуальные методы — hook-и), паттерн “Наблюдатель”, “дедушка всех паттернов” MVC. Такой подход несколько затрудняет отладку, но позволяет разрабатывать каждый уровень независимо, и не думать, как будут пользоваться кодом уровня другие.

В реальной жизни часто встречается диаметральная противоположность такого подхода — антипаттерн “умный GUI”. Это когда вся бизнес-логика приложения реализуется прямо в классах, отвечающих за пользовательский интерфейс, как правило, в обработчиках библиотечных событий. При этом даже работа с БД может выполняться из кода GUI.

Как ни странно, это не всегда плохо. Во-первых, есть куча тулов для быстрой разработки пользовательских интерфейсов, при этом подходе их можно задействовать на полную — большая часть приложения будет просто сгенерена. Во-вторых, это быстро — рабочее приложение таким способом можно написать за пару часов, при этом нет оверхеда на создание классов, интерфейсов, наладку общения между уровнями, организацию взаимодействия. Легко добавлять в приложение новые фичи — просто кидаем на форму но-

вый контрол, цепляем к нему новый обработчик, и в продакшн. Причём, поскольку код каждого куска такой функциональности довольно прост, его несложно поддерживать или переписать заново.

Однако если ожидаемый размер проекта превышает где-то 3000 строк кода, “Умный GUI” быстро становится антипаттерном. Почему: невозможно проектирование по модели, классов предметной области просто нет, есть методы-обработчики, в которых код интерфейса смешан с бизнес-правилами и инфраструктурным кодом. Поэтому переиспользование бизнес-правил чрезвычайно затруднено, фактически, если две фичи требуют одного бизнес-правила, его надо реализовать дважды. Сложное поведение в такой ситуации оказывается сложно реализовать — не получится аккуратно декомпозировать его на набор взаимодействующих объектов без разделения на уровни. Практически невозможна интеграция с другими системами — всё завязано на GUI, так что реализовать отдельный API для интеграции может быть очень сложно (опять-таки, в процессе сами собой появятся бизнес-объекты и уровни, так почему не сделать сразу нормально?).

## 5. Основные структурные элементы модели

В программе модель реализуется с помощью различных “элементарных” блоков, и не всегда тривиально. В первую очередь, следует подумать над ассоциациями. Если в предметной области одно из направлений ассоциации более приоритетно, чем другое (например, “страна – президент”, скорее всего, предполагает доступ от страны к президенту), то имеет смысл сделать ассоциацию односторонней. В идеале все или почти все ассоциации должны быть односторонними, потому что двунаправленная ассоциация делает невозможным понять один объект в отрыве от другого. Также следует по возможности минимизировать множественность (возможно, добавив квалификаторы, типа “страна – год – президент”) и минимизировать количество ассоциаций вообще. Меньше связей — проще анализ и сопровождение.

Следующая задача — решить, кто в модели будет сущностью, а кто — объектом-значением. Сущность имеет собственную идентичность, не зависящую от её состояния, тогда как объект-значение определяется только значениями своих атрибутов. Это всё очень похоже на ссылочные типы и типы-значения в языках программирования, но в реальной жизни всё гораздо сложнее, потому как ссылочная идентичность не переживает сериализации. Кроме того, одна и та же сущность может вообще представляться несколькими совершенно разными объектами (например, когда ввод данных в систему происходит в нескольких приложениях), или существовать в нескольких экземплярах в системе (например, в случае с распределёнными транзакциями). Тем не менее, система в любом случае должна корректно устанавливать идентичность сущности. Простой пример возможных проблем — входящий звонок от клиента. Это может быть старый клиент, звонящий с известного нам номера, либо старый клиент, который поменял себе номер (и мы всё равно должны как-то догадаться, что уже имели с ним дело), или даже новый клиент, которому почему-то достался номер старого клиента (и мы должны догадаться, что это новый клиент).

В реальной жизни естественные механизмы обеспечения идентичности не очень-то существуют. Людей, например, можно было бы идентифицировать по номеру паспорта, но паспорта меняют. Можно было бы по ИНН, но не у всех он есть. Можно было бы по СНИЛС, но он тоже есть не у всех. Имя и фамилия тем более не лучший способ иденти-

ификации, потому как даже я учился на одном потоке с человеком, у которого полностью совпадали со мной имя, фамилия и первая буква отчества (и год рождения, раз мы были на одном потоке). И нас правда часто путают до сих пор, меня уже несколько раз приглашали работать Java-программистом (один раз в Яндекс). Эрик Эванс в своей книге рассказывал гораздо более грустную, но аналогичную историю.

Часто в качестве идентификатора используют некий суррогатный атрибут, например, GUID, назначаемый объекту при создании. Иногда этот атрибут даже становится доступен пользователю (например, трек-номер почтового отправления), иногда нет (например, у каждого документа в Google Docs есть уникальный Id, который нигде в интерфейсе не показан, поэтому в одной папке вполне может быть два документа с одним именем). Иногда есть естественный атрибут, однозначно идентифицирующий сущность, например, номер кресла в кинотеатре. Да, кресло могут переставить на другое место или сменить нумерацию, но для информационной системы кинотеатра важна не физическая идентичность кресла, а место, билет на которое можно продать. Естественные атрибуты лучше суррогатных, поскольку сами собой решают проблему идентификации “дубликатов” объекта (той же сущности, но из другой системы), но пользоваться ими надо очень осторожно, потому что то, что вам кажется уникальным и присущим всем объектам свойством, на деле может оказаться не так.

От способа идентификации требуется, чтобы он переживал сохранение, загрузку, передачу в другую систему и представление там в другом формате. Соответственно, нужна операция идентификации, которая по двум данным объектам может сказать, один и тот же это объект или нет. Ссылочное равенство, очевидно, не подходит для большинства практических полезных случаев, поэтому способ идентификации на самом деле важное архитектурное решение.

Можно всё в системе сделать сущностями, но тогда все рассуждения про сложные правила идентификации, приведённые выше, оказываются применимы к вообще каждому объекту системы, что, во-первых, доставит боль при реализации, а во-вторых, повредит эффективности. Поэтому всё, что можно, лучше превращать в объекты-значения. Объекты-значения должны быть единым концептуальным целым (например, имя и фамилия — это не два атрибута сущности “Человек”, а один объект-значение, описывающий человека), очень желательно делать их немутабельными. Объекты-значения вполне могут ссылаться на сущности, и наоборот, сущности могут ссылаться на объекты-значения. Обратите внимание, что объекты-значения может быть вполне валидно реализовывать как ссылочные типы (например, чтобы обеспечить разделяемость). Немутабельность, кстати, полезна не только для обеспечения разделяемости, а ещё и для того, чтобы иметь возможность возвращать или передавать как параметр объект-значение — мы отдаём его вовне, но точно знаем, что его никто не сломает. Разделляемость, кстати, хоть и хороша в плане экономии памяти и места в базе данных, может быть плоха, если система распределённая — при каждом обращении к разделяемому объекту придётся делать сетевой запрос.

Обратите внимание, что сущность объект или значение — это не его врождённое свойство, а вопрос проектирования конкретной системы. Один и тот же объект может играть роль как сущности, так и значения, даже в рамках одной предметной области. Например, бывают билеты на места с указанием места (и тогда каждое место — сущность, у него есть идентичность, исключающая возможность продать два билета на одно место), а бывают билеты без указания места (и тогда важно только количество мест, оно будет объектом-значением). В последнем случае считать место сущностью будет даже неправильным (хоть

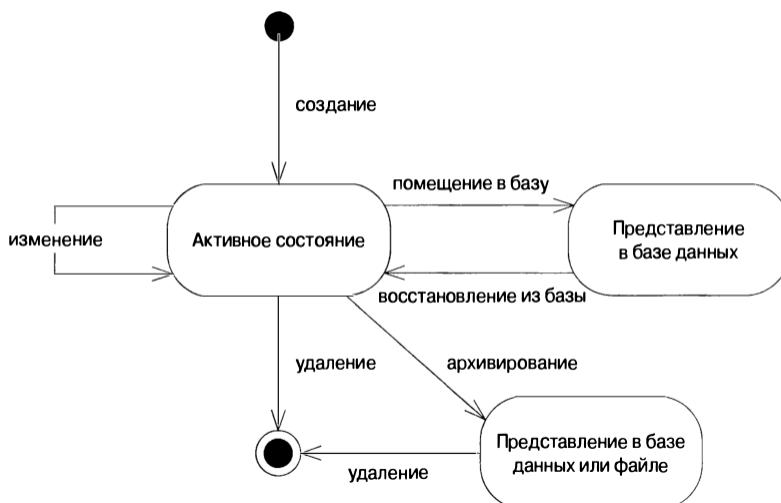
на месте всё ещё написан его номер), поскольку в разы без всякой нужды усложняет систему.

В DDD разрешено и даже поощряется то, что считается дурным тоном в ООП — классы без состояния, фактически представляющие собой набор статических методов. В DDD их называют “службами”. В службы выносится код, который не имеет естественного места в “настоящих объектах” — часто это операция, выполняемая над объектами нескольких разных классов, где все объекты равноправны, поэтому нельзя естественным образом поместить эту операцию в один из классов-участников — например, различного рода диспетчеры или менеджеры. Службам, как правило, запрещается иметь состояние, так что вызывать методы службы можно отовсюду и в любом порядке, не заботясь о порядке вызовов.

Последней элементарной частью модели является модуль. Да, это модули в обычном для языков программирования смысле, DDD лишь призывает использовать их не как техническое средство группировки классов, а как реальный механизм декомпозиции предметной области. Классы следует объединять в модули не по принципу высокой внутренней связности (*cohesion*) и низкой внешней зависимости (*coupling*), а семантически. Модуль — это важная часть модели, модули должны быть определены так, чтобы повышать “объясняющую способность” модели предметной области. А высокая связность и низкая зависимость при этом получатся сами собой. Если это не так, имеет смысл порефакторить модель. Имена модулей должны естественным образом вписываться в единый язык модели.

## 6. Цикл существования объекта модели

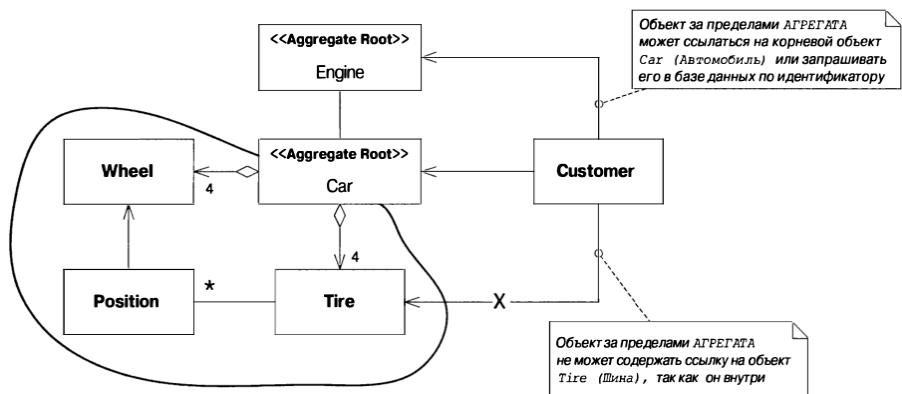
Объекты в информационных системах часто обладают несколько более сложным жизненным циклом, чем “создали, вызвав конструктор -> попользовались -> удалили”. Общий вид цикла существования объекта модели предметной области такой:



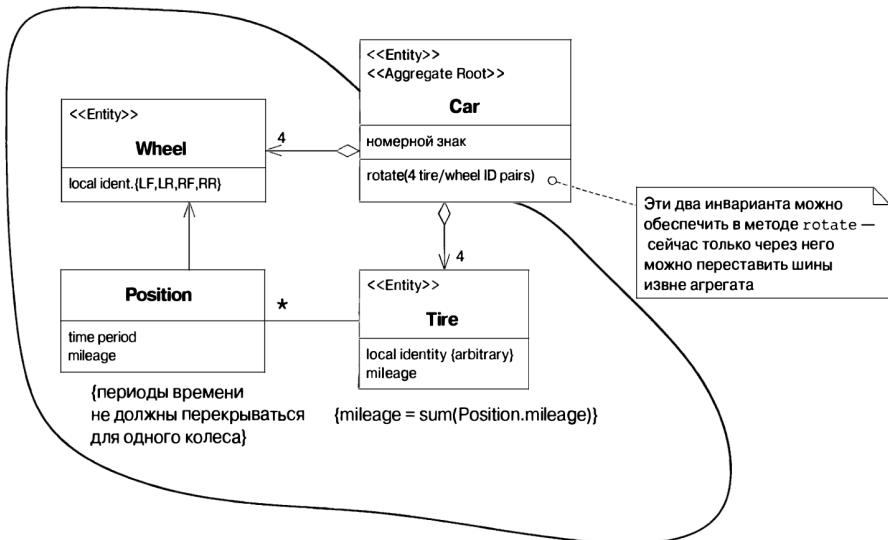
Тут, как обычно, возникают проблемы с идентичностью объекта, если это сущность, а не значение, но кроме этого, добавляются проблемы с целостностью объекта и с излишней сложностью управления его состоянием. Для решения этих проблем применяются некоторые широко известные паттерны проектирования.

Первый паттерн — паттерн “Агрегат”. Агрегат — это изолированный кусок модели, имеющий “корень” и “границу”. Доступ ко всем элементам агрегата возможен только через его корень. Корень представляет собой объект-сущность и имеет свою идентичность в рамках всего приложения. Остальные объекты в агрегате могут быть как сущностями, так и значениями, и они должны поддерживать свою идентичность только в рамках агрегата. Для чего это нужно — для упрощения и декомпозиции модели. Когда мы пишем агрегат, мы можем думать только о взаимосвязи объектов внутри агрегата, а когда мы пишем код, пользующийся агрегатом, мы можем думать только о его корне. Так что агрегат — это такая единица инкапсуляции программы. Как класс, только включает в себя, возможно, несколько классов. С технической точки зрения аккуратное разделение на агрегаты в разы облегчает обновления в базе данных: транзакции, затрагивающие разные агрегаты, гарантированно не мешают друг другу.

Содержимое агрегата на самом деле может быть видно снаружи, и не возбраняется отдавать ссылки на внутренние объекты агрегата, просто извне их нельзя хранить. Кроме того, у агрегата (как у класса) есть свои инварианты, и за поддержание их отвечает корень. Поэтому объекты, которые отдаются вовне, скорее всего, должны быть немутабельны, или быть копиями объектов внутри агрегата. Вот небольшой пример:



Тут машина — агрегат, содержащий в себе колёса и шины, поскольку пользователю может быть важно, на каком колесе какая шина, но глобально идентифицировать шины не требуется. Двигатель же наоборот, хоть и является неотъемлемой частью автомобиля, имеет собственный номер, так что уникален глобально (и на самом деле номер двигателя иногда проверяют безотносительно номера автомобиля). Автомобиль же предоставляет всю функциональность для управления шинами извне и поддерживает свои инварианты:



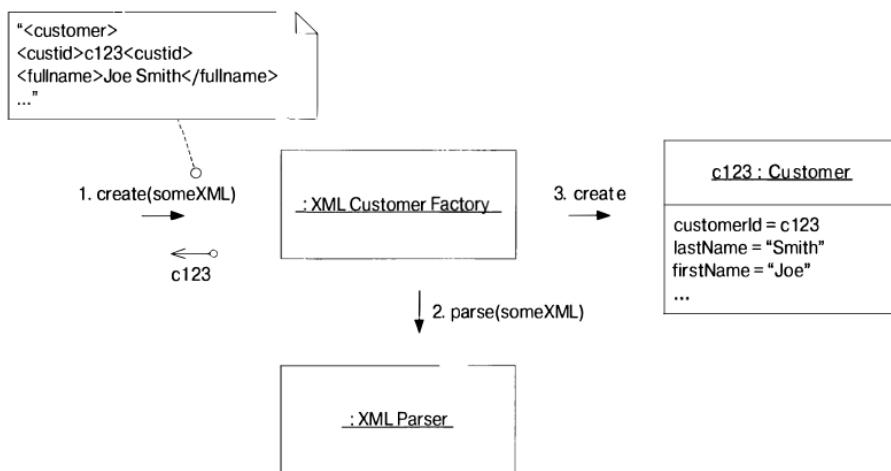
Следующий паттерн — “Фабрика”. В DDD это несколько более архитектурный паттерн, чем похожий паттерн “Абстрактная Фабрика” Банды Четырёх, и на самом деле может использовать абстрактную фабрику для реализации. Задача фабрики — создать и проинициализировать агрегат (или даже просто один объект) так, чтобы выполнялись его инварианты. Фабрика инкапсулирует сложную операцию создания объекта, не раскрывая его внутреннюю структуру, и при этом избавляет сам агрегат от необходимости уметь создавать себя. Фабрика вправе знать о структуре агрегата и манипулировать его внутренним содержимым непосредственно, а код извне — нет, так что без фабрики пользоваться агрегатом было бы зачастую невозможно:



Клиент передаёт фабрике параметры, указывающие, что ему нужно (может использоваться паттерн “Спецификация”, о котором чуть потом), фабрика создаёт продукт и возвращает клиенту результат. Процесс создания атомарен в том смысле, что если что-то пошло не так, то клиент не получит результата вообще, как будто ничего не было. Для реализации фабрики могут использоваться паттерны “Фабричный Метод”, “Абстрактная Фабрика”, “Builder”.

Интересно, что фабрики очень редко присутствуют явно в предметной области, тем не менее, являются важной частью модели и должны учитываться и в модели, и в едином языке на равных правах с “настоящими” объектами.

Фабрики могут использоваться не только для создания, но и для восстановления объекта из внешнего хранилища. В этом случае они не должны присваивать объекту новый идентификатор или вообще как-либо портить его идентичность. Вот пример такой фабрики, основная работа которой заключается даже не в создании объекта, а на самом деле в разборе XML-ки, куда был сохранён объект:



Следующий паттерн управления жизненным циклом — это “Репозиторий” (или “Хранилище”). Репозиторий отвечает за хранение, сохранение и загрузку объектов при необходимости, предоставляя, как правило, глобальный доступ к объектам. Чаще всего репозиторий представляет собой прослойку между приложением и базой данных, при этом репозиторий сам может держать объекты в памяти и возвращать их по запросу. Нужно это для того, чтобы минимизировать количество переходов по ссылке с целью найти какой-либо объект, и вообще упростить задачу поиска. Представьте себе, что было бы, если бы все объекты в программе можно было найти только переходом по ссылкам от некоторого корневого объекта. Если задача такая, что всё всегда помещается в память и данных очень немного, то всё было бы и так хорошо, но если большая часть данных всё время хранится в БД, то такой поиск блокировал бы половину базы, делая практически невозможной параллельную работу. Часто лучше хранить Id-шник нужного объекта и получать его при необходимости из репозитория, чем держать ссылку на объект.

Репозиторий может использовать развитый язык запросов, чтобы позволять клиенту найти объект. Часто это просто какой-то метод идентификации (типа суррогатного Id), но вполне может быть выборка объекта или коллекции объектов по их атрибутам или каким-то другим признакам (например, участии в отношениях с другими объектами). Опять-таки, может использоваться паттерн “Спецификация”, про который чуть позже. Репозиторий за интерфейсом запросов прячет метод реального получения объекта: внутри он может использовать ORM-библиотеку + реляционную БД, может объектно-ориентированную БД, может (и часто использует) фабрики для создания объектов или восстановления объектов из базы. Задача репозитория — делать так, чтобы обо всём этом не надо было заботиться клиентскому коду:



Репозитории хорошо дружат с агрегатами, предоставляя доступ к корневому объекту агрегата по глобальному Id и собирая остальной агрегат при необходимости.

## 7. Пример, система грузоперевозок

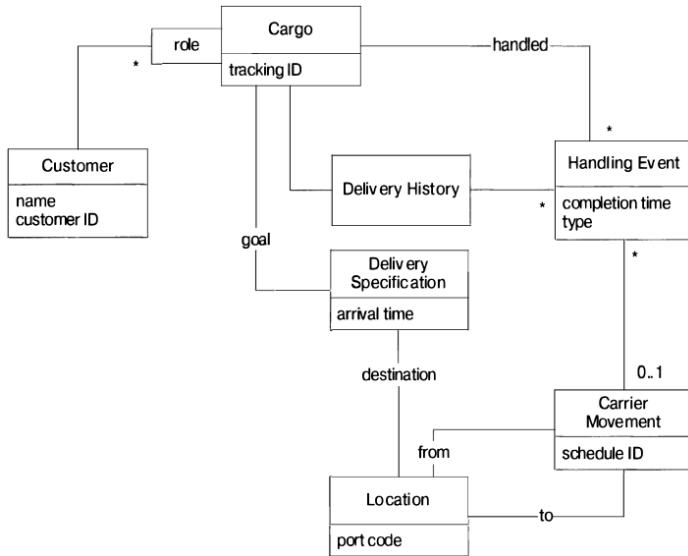
Рассмотрим большой пример, который с нуля попробуем спроектировать по принципам DDD и с использованием рассмотренных паттернов. Требуется разработать информационную систему для логистической компании, которая могла бы помогать принимать заказы на перевозку грузов, формировать маршрут перевозки и отслеживать движение груза по этому маршруту. Более подробно, вот первый набросок требований:

1. Отслеживать ключевые манипуляции с грузом клиента
2. Оформлять заказ заранее
3. Автоматически высыпать клиенту счет-фактуру по достижении грузом некоторого операционного пункта маршрута

Первая задача при проектировании — выделить сущности предметной области. Для этого требуется проинтервьюировать экспертов и узнать, что:

- В работе с Грузом (Cargo) участвует несколько Клиентов (Customers), каждый из которых играет свою роль (Role)
- Должна задаватьсяся (be specified) цель (goal) доставки груза
- Цель (goal) доставки груза достигается в результате последовательности Переездов (Carrier Movement), которые удовлетворяют Заданию (Specification)

Начинает вырисовываться единый язык, в котором пока только термины исключительно из предметной области. Нарисуем первое приближение модели в виде диаграммы классов UML:



В центре системы находится Груз (Cargo), который нужно доставить в указанный порт не позднее указанного времени. Порт и время указываются в Delivery Specification, которой должен удовлетворять маршрут. Сама перевозка состоит из Handling Event-ов, связанных в Delivery History для каждого груза. Сам Handling Event имеет ссылку на соответствующий ему Carrier Movement, который, в свою очередь имеет место отправления и место прибытия. При этом есть ещё Customer, который может следить за своим грузом, имея его tracking ID. Обратим внимание, что Carrier Movement может за один раз перемещать сразу много грузов, что у Customer-а может быть много грузов, что Handling Event относится всегда к одному грузу, но в процессе доставки их, естественно, может быть много.

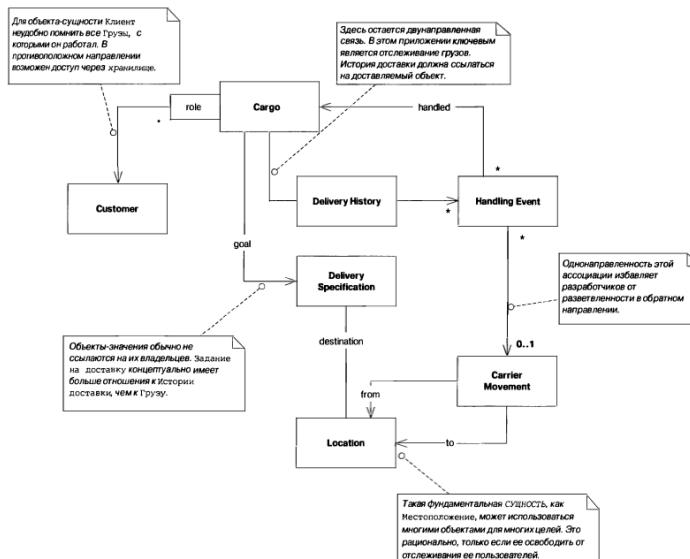
Какой-то набросок модели предметной области есть, дальше определимся с высокоД уровневой структурой приложения. Будем использовать уровневую архитектуру (как это принято), с уровнем пользовательского интерфейса, операционным, предметной области и уровнем утилит. Операционный уровень (или уровень приложения) сейчас нам наиболее интересен, поскольку позволит определиться с требуемой функциональностью, а понимая её мы сможем целенаправленно уточнять модель. Судя по ТЗ и общему пониманию задачи, нам потребуются три разных приложения.

- Маршрутный запрос (Tracking Query) — доступ к прошлым и нынешним манипуляциям с конкретным грузом, будет использоваться клиентом, чтобы следить за грузом.
- Служба резервирования (Booking Application) — позволяет заказать доставку нового груза, будет использоваться клиентом.
- Служба регистрации событий (Incident Logging Application) — регистрирует действия с грузом, будет использоваться сотрудниками логистической компании, чтобы вводить данные, которые потом будут показывать Tracking Query.

Теперь можно заняться уточнением модели. Определимся, кто из объектов модели сущность, а кто может быть объектом-значением:

- **Клиент (Customer)** — сущность, у него точно есть идентичность;
- **Груз (Cargo)** — сущность, тоже, идентичность следует из желания клиента знать, что с его грузом;
- **Манипуляция (Handling Event) и Переезд (Carrier Movement)** — более сложно, но, в общем-то, в реальной жизни это уникальные операции, каждая из которых отличается от другой, и это важно для приложения, отслеживающего грузы — значит, пусть будут сущностями;
- **Местоположение (Location)** — сущность, нам надо различать два порта даже с одинаковым названием. Можно было бы использовать координаты, но это на самом деле не важно и не очень удобно, так что пусть будет сущность с искусственным механизмом идентификации (код порта);
- **История доставки (Delivery History)** — сущность, локально идентичная в пределах агрегата “Груз” — сама по себе история доставки уникальна и не взаимозаменяется с другими, но интересна только для конкретного груза;
- **Задание на доставку (Delivery Specification)** — значение, потому как это просто задание, а не конкретная доставка. Его вполне можно переиспользовать, и два задания, требующие одного и того же — по сути, одно задание.
- Всё остальное — значения.

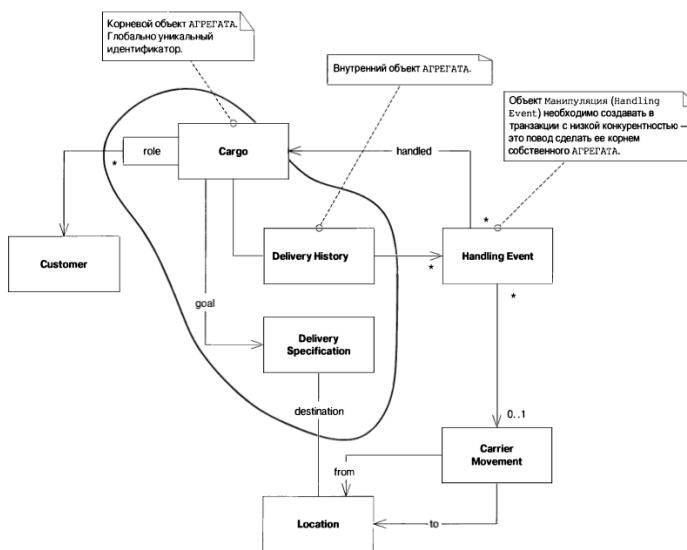
Теперь можно определиться с направленностью ассоциаций:



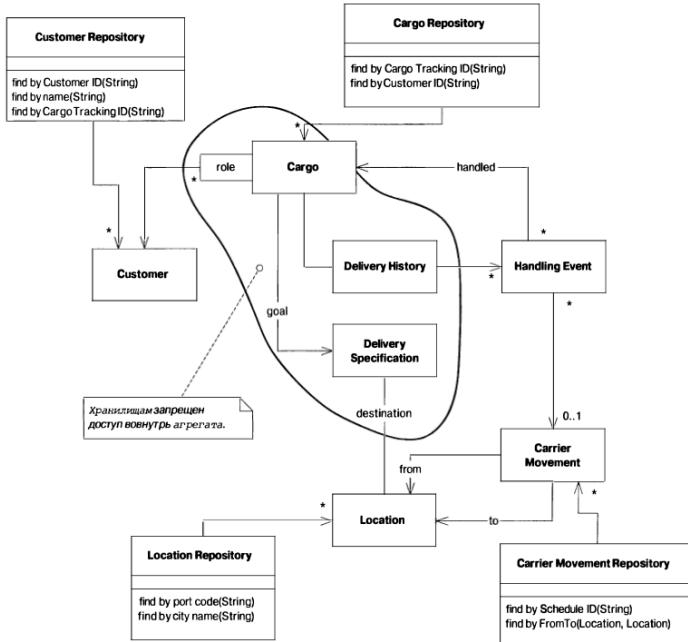
Мы постараемся минимизировать количество ссылок за счёт использования паттерна “Хранилище”. Для начала разрешим Клиенту не ссылаться на Груз — для постоянных

клиентов, которые отправляют тысячи грузов, держать их все в памяти неудобно, поэтому можно сделать Грузу ссылку на Клиент и в обратную сторону получать информацию с помощью запроса в БД, через “Хранилище”. Дальше сделаем ассоциацию между Handling Event и Carrier Movement односторонней, от первого ко второму — мы не хотим вести учёт грузовых кораблей, мы хотим вести учёт грузов, поэтому какие грузы везёт конкретный корабль — не так важно, это можно узнать снова запросом к БД через репозиторий. Связи с участием Location должны быть все односторонними, причём в сторону Location, потому как местоположение используется много где и много ком, и ему вовсе не обязательно знать, кем именно. Между грузом и историей доставки связь всё-так будет двунаправленной, и будет циклическая зависимость между грузом, историей доставки и манипуляцией, но эта зависимость, судя по всему, присуща предметной области и от неё в модели особо никуда не деться. В реализации же можно заменить одну из ассоциаций на запрос к репозиторию.

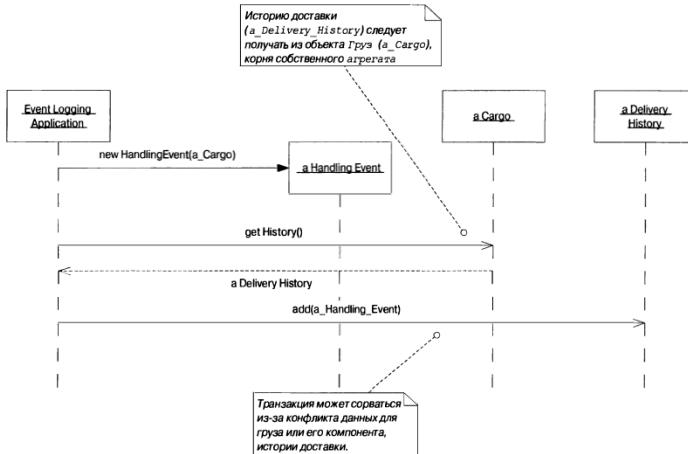
Дальше определимся с агрегатами. Вокруг Груза есть некоторое количество локально идентичных объектов, поэтому естественно было бы сделать Груз корнем агрегата. А вот Handling Event, несмотря на то, что кажется логичным включить его в Груз, используется и сотрудниками логистической компании, чтобы отмечать статус груза, и тогда Handling Event надо уметь находить по Перевозке (приплыл корабль с сотней разных грузов — надо все пометить как доставленные). Поэтому его разумно сделать корнем своего агрегата, не зависящего от Груза:



Последнее, что надо решить — какие репозитории будут использоваться в системе. Надо, очевидно, уметь находить груз по Клиенту, Клиента по имени и по грузу, Location по имени города или по идентификатору порта, Перевозку (опять-таки, приплыл корабль — нашли его в базе и запросили все Handling Event-ы, с ним связанные). Handling Event-у репозиторий пока не нужен, потому что его всегда можно найти по истории Груза, а требований, касающихся отслеживания грузов, которые перевозит корабль, пока нет. Поэтому пока получилось как-то так:



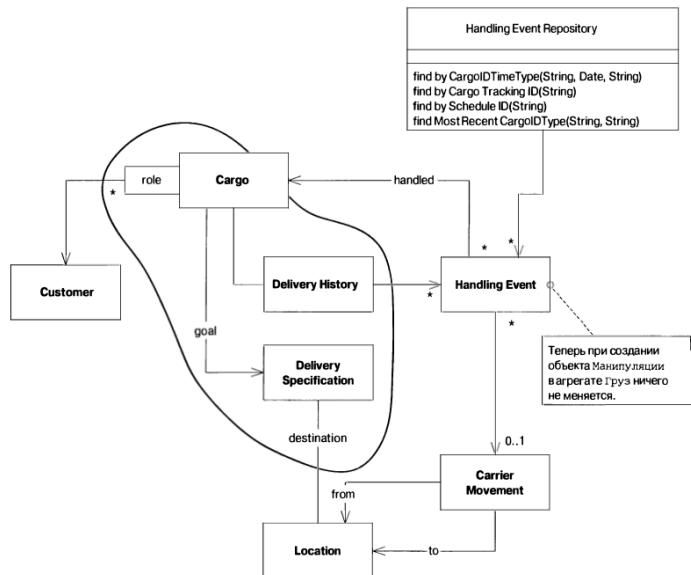
Опробуем получившуюся модель, пройдясь по одному из случаев использования — добавлению события манипуляции с грузом сотрудником логистической компании в порту. Поскольку мы решили не делать репозиторий для Handling Event-ов и ассоциация от Перевозки к Handling Event-у у нас односторонняя, то получится как-то так:



Получилось не очень удобно, запрос делается весьма окольными путями, да ещё и блокирует половину базы данных в процессе (нам требуется весь агрегат “Груз”, чтобы добавить событие в его историю).

Поэтому необходим рефакторинг, который исправил бы последствия допущенной нами ошибки. Давайте всё-таки сделаем Handling Event-ам свой репозиторий, и не будем

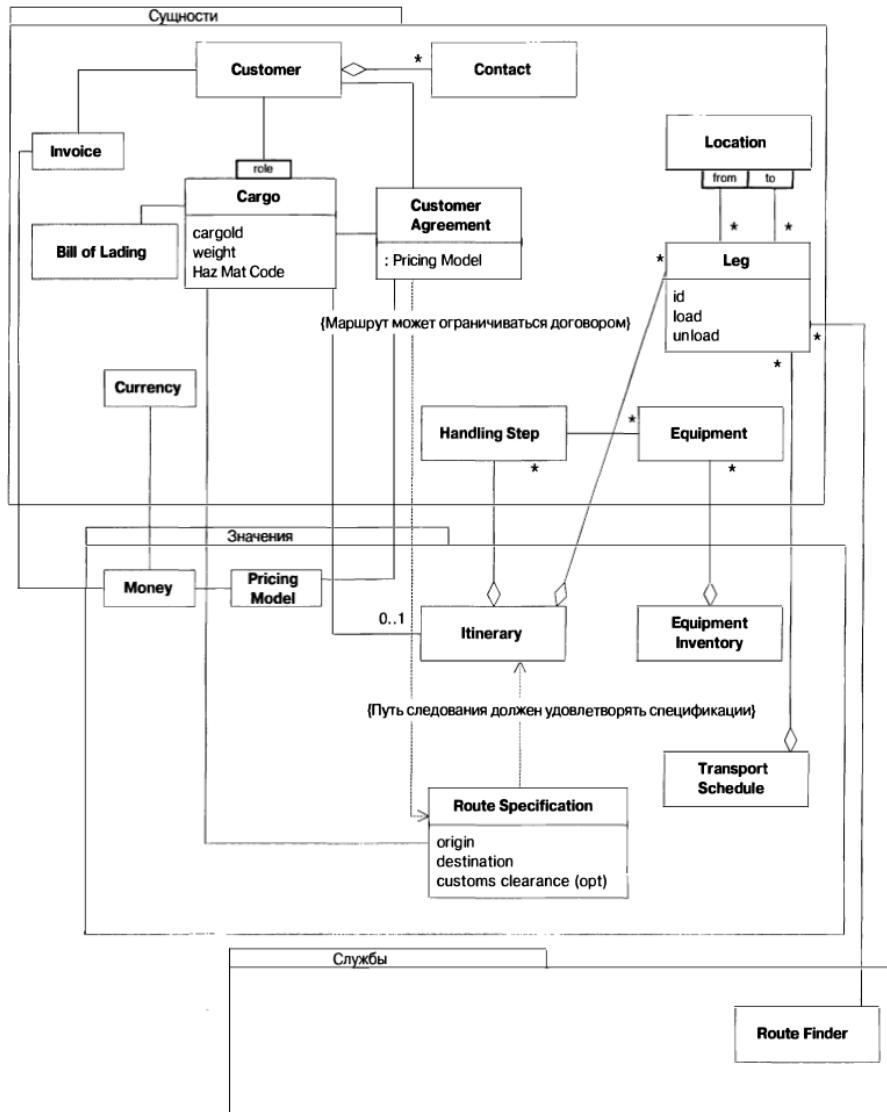
ссылаясь на Handling Event из Delivery History напрямую. Ссылку теперь, раз есть репозиторий, можно заменить на запрос к базе:



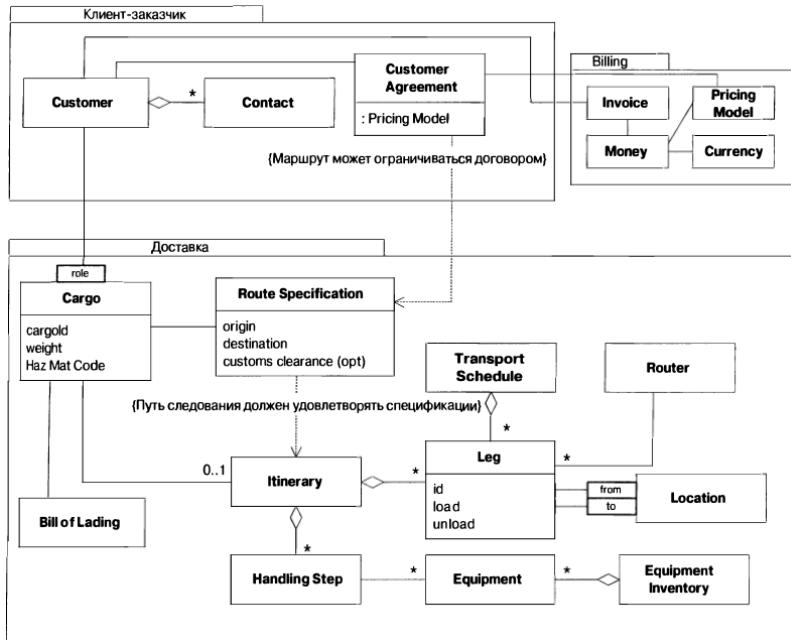
Стало лучше, теперь при добавлении нового события в агрегате “Груз” ничего не меняется. В концептуальной модели ассоциация между Delivery History и Handling Event есть, но в коде она реализуется как запрос к репозиторию.

## 8. Разбиение по модулям

Пример с грузоперевозками понадобится нам ещё для того, чтобы обсудить важный принцип DDD о том, что разбиение на модули должно выполняться не механически (например, по типам), а семантически, с учётом требований предметной области. Представим себе, что проектирование системы грузоперевозок дошло до стадии, когда объектов стало слишком много и потребовались модули для декомпозиции системы. Можно сделать плохо, разделив все классы по модулям на основании их типов — сущности, значения или службы:



А можно — хорошо, в соответствии с единым языком. Программа работает с Клиентом, выполняя Доставку, за что берёт с него Деньги:

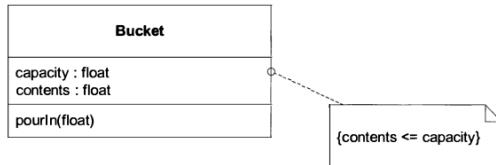


Как видим, при таком разделении сопряжение и связность тоже стали лучше. На самом деле, это не всегда так, иногда чисто механическая гонка за low coupling и high cohesion ухудшает модель, а не улучшает её. Главное в модели — это не метрики, а её объясняющая способность.

## 9. Моделирование ограничений

Отвлечёмся от примера про систему грузоперевозок и рассмотрим такой важный момент проектирования, как моделирование и реализацию ограничений на состояние системы. Как мы увидим, ограничения могут привести к внезапному появлению новых объектов в модели, которых как будто не было в предметной области.

Рассмотрим простой пример — ведро. В него можно налить. Но не больше, чем в него помещается. Такие дела.



Как это можно было бы реализовать на Java:

```
class Bucket {
    private float capacity;
```

```

private float contents;

public void pourIn(float addedVolume) {
    if (contents + addedVolume > capacity) {
        contents = capacity;
    } else {
        contents = contents + addedVolume;
    }
}

```

На самом деле, так себе — код проверки ограничения смешан с кодом, делающим полезную работу, и несмотря на то, что тут и так всё просто, можно сделать лучше, вынеся проверку ограничения явно:

```

class Bucket {
    private float capacity;
    private float contents;

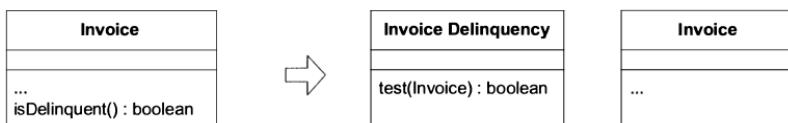
    public void pourIn(float addedVolume) {
        float volumePresent = contents + addedVolume;
        contents = constrainedToCapacity(volumePresent);
    }

    private float constrainedToCapacity(float volumePlacedIn) {
        if (volumePlacedIn > capacity) return capacity;
        return volumePlacedIn;
    }
}

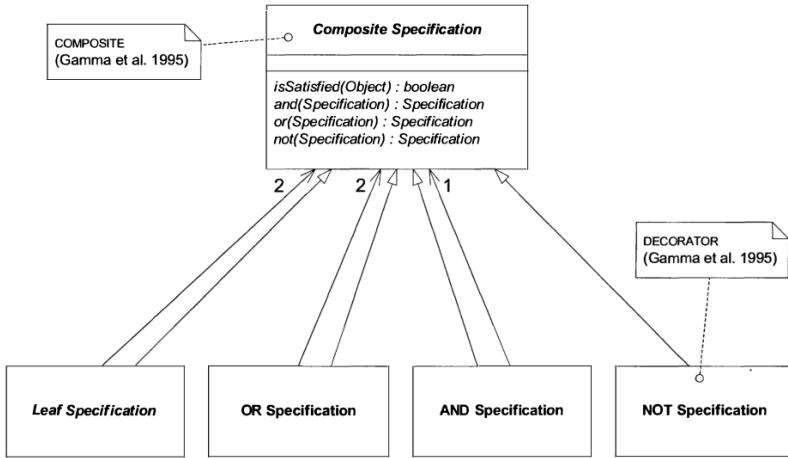
```

Теперь у нас есть метод, у которого есть говорящее имя, по которому сразу понятно, что происходит, и ограничение уже не потерять.

Однако более сложным ограничениям и в отдельном методе становится тесно. Тогда имеет смысл создать отдельный класс, работа которого будет состоять в проверке объекта на соответствие некоторому ограничению. Как правило, такой класс содержит в себе булевый метод, который говорит, удовлетворяет объект спецификации или нет:

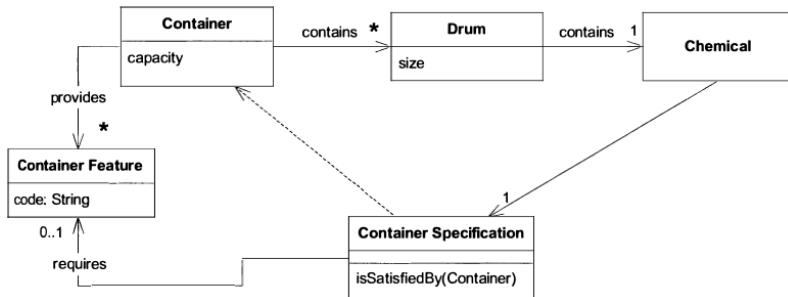


Особенно хорошо то, что спецификация хорошо работает с паттерном “Компоновщик”, что позволяет создавать сложные спецификации, комбинируя более простые:



Спецификация полезна не только для проверки ограничений как большой и страшный assert, но, и прежде всего, как инструмент передачи требований при выборке объекта или при его конструировании в фабрике. Спецификация позволяет сказать, например, “Хочу граф, односторонний, где, скорее всего, будет примерно столько же вершин, сколько и ребёр”, и по этой спецификации фабрика может выбрать наиболее подходящую реализацию графа. В более распространённом для информационных систем случаев спецификация может представлять собой что-то вроде дерева разбора оператора SQL, скорее всего, сильно упрощённого, чтобы ей всё ещё было удобно пользоваться.

Небольшой пример того, как это выглядит в коде. Положим, у нас есть склад химикатов, где химикаты хранятся в бочках (Drum) внутри контейнеров. Химикаты бывают обычные, легко испаряющиеся и взрывоопасные, соответственно, и хранить их надо либо в обычных контейнерах, либо в контейнерах с вентиляцией, либо в бронированных контейнерах. Обычные химикаты можно хранить где угодно, но мы хотим добиться того, чтобы место в контейнерах использовалось наиболее эффективно. Модель предметной области могла бы быть такой:



Каждый контейнер обеспечивает сколько-то характеристик (container feature), и требования химиката касательно условий хранения можно записать как спецификацию, которой может удовлетворять контейнер. У каждого химиката есть ссылка на спецификацию, спецификация может ссылаться на характеристику контейнера и проверять контейнер на то, обладает он характеристикой или нет. В коде это выглядит очень просто:

```

public class ContainerSpecification {
    private ContainerFeature requiredFeature;

    public ContainerSpecification(ContainerFeature required) {
        requiredFeature = required;
    }

    boolean isSatisfiedBy(Container aContainer) {
        return aContainer.getFeatures().contains(requiredFeature);
    }
}

```

Ну и в классе Контейнер можно воспользоваться спецификацией, чтобы проверить, правильно ли он заполнен:

```

boolean isSafelyPacked() {
    Iterator it = contents.iterator();
    while (it.hasNext()) {
        Drum drum = (Drum) it.next();
        if (!drum.containerSpecification().isSatisfiedBy(this))
            return false;
    }
    return true;
}

```

Как видим, получилось довольно объектно-ориентированно и аккуратно.

## 10. Гибкая архитектура

Ну и последнее, про что следует рассказать, говоря про “тактические” аспекты DDD — это приёмы обеспечения “гибкости” архитектуры, то есть набор несложных правил кодирования, которые сделают рефакторинг (в том числе архитектурный) гораздо более безболезненным.

Рассмотрим для примера задачу расчёта смешения красок в строительном магазине. Есть краска, характеризующаяся своим объёмом и цветом (в шкале RGB), надо по данным двум краскам посчитать, какой краской будет результат их слияния. Вот исходное состояние нашей системы:

Paint
v : double
r : int
y : int
b : int
paint(Paint)

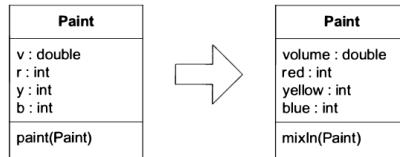
```

public void paint (Paint paint) {
    v = v + paint.getV(); // После смешивания объем суммируется
}

```

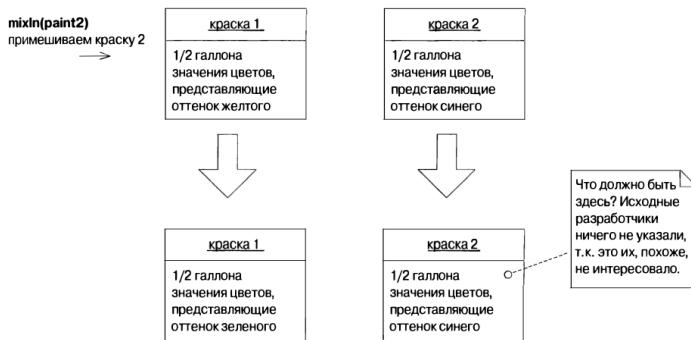
```
// Опущено много строк сложного расчета смешивания цветов,
// который заканчивается присваиванием новых значений
// компонентов r (красного), b (синего) и у (желтого).
}
```

Первое правило заключается в том, что, ГМ, надо делать хорошо и не надо делать плохо. Более конкретно, имена, используемые в программе, должны соответствовать предметной области и объяснять происходящее в коде:



```
public void testPaint() {
    // Начинаем с чистой желтой краски объемом = 100
    Paint ourPaint = new Paint(100.0, 0, 50, 0);
    // Берем чистую синюю краску объемом = 100
    Paint blue = new Paint(100.0, 0, 0, 50);
    // Примешиваем синюю краску к желтой
    ourPaint.mixIn(blue);
    // Должно получиться 200.0 единиц зеленой краски
    assertEquals(200.0, ourPaint.getVolume(), 0.01);
    assertEquals(25, ourPaint.getBlue());
    assertEquals(25, ourPaint.getYellow());
    assertEquals(0, ourPaint.getRed());
}
```

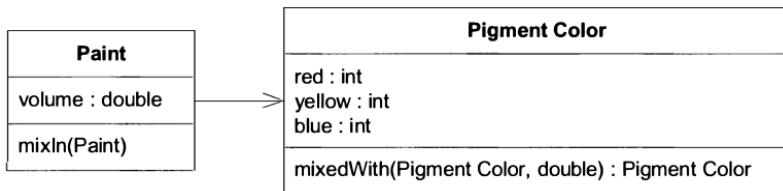
Однако более интересное правило — правило избегания побочных эффектов и активного использования немутабельных объектов. Проблема:



Программа работает правильно, просто она непонятна. А мы знаем, что хорошая модель (и, следовательно, хорошее её выражение в коде) должно прежде всего объяснять.

Существующая модель ничего не говорит о том, что будет со второй краской, у вдумчивого разработчика появляются вопросы, а это как раз то, чего модель должна помогать избегать — она должна отвечать на вопросы, а не ставить их.

Поэтому посмотрим на смешивание красок по-другому. Выделим сущность “цвет”, и сделаем краску сущностью, которая обладает этим самым цветом и объёмом. Ещё сделаем наблюдение, что цвет результата на самом деле зависит не от объёма, а от процентного соотношения объёмов смешиваемых цветов. Получаем вот такую модель:



И вот такой код:

```
public class PigmentColor {
    public PigmentColor mixedWith(PigmentColor other, double ratio) {
        // Много строк сложного расчета смешивания цветов.
        // в результате создается новый объект PigmentColor
        // с новыми пропорциями красного, синего и желтого.
    }
}

public class Paint {
    public void mixIn(Paint other) {
        volume = volume + other.getVolume();
        double ratio = other.getVolume() / volume;
        pigmentColor = pigmentColor.mixedWith(other.pigmentColor(), ratio);
    }
}
```

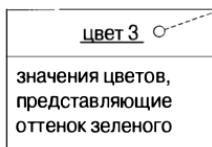
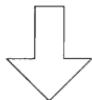
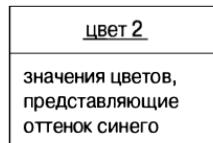
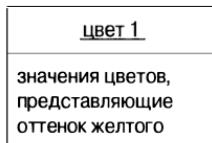
Теперь вместо того, чтобы думать, что произойдёт с примешиваемой краской, мы определяем смешиванием цветов, результатом которого станет новый цвет, и оба старых цвета гарантированно не изменятся:

**mixedWith(color2)**

смешиваем с цветом 2



color 3



Создан новый  
ОБЪЕКТ-ЗНАЧЕНИЕ.  
Прежние объекты  
не изменяются.

Осталось только разобраться с объёмами, в самой краске, потому как цвет про объём теперь ничего не знает.

Сейчас получается довольно-таки странно:

Постусловие для mixIn():

После `pl.mixIn(p2):`

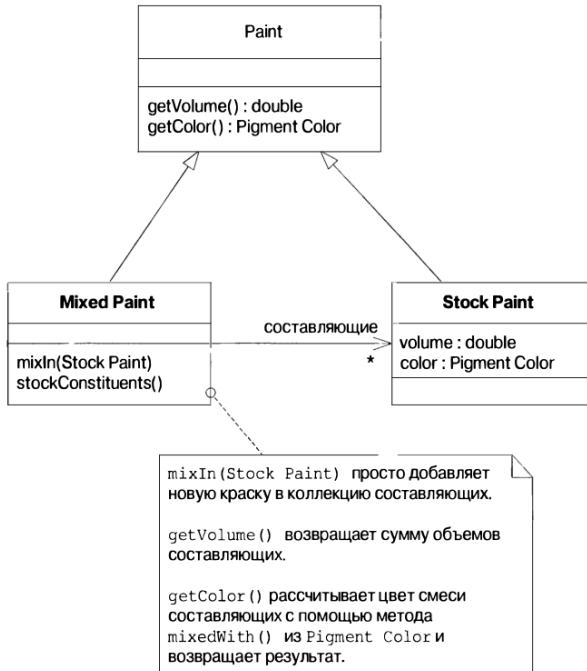
`pl.volume` увеличивается на объем `p2.volume`

`p2.volume` не изменяется

И инвариант:

Общий объем краски не должен измениться от смешивания

Пока что модель вообще противоречива, её объясняющая способность нулевая. Зарефакторим её. Разделим понятия “краска” и “смесь красок”, так что смесь можно формировать только из заранее известного набора базовых красок, которые берутся с заданным объёмом. И никакого смешения красок каждый раз на самом деле не происходит, смесь просто хранит в себе список базовых красок с объёмами, и при необходимости вычисляет получившийся цвет. Получилось как-то так:



Понятно, что это не единственное возможное решение, но оно вполне работоспособно, убирает возможные противоречия из модели и даже соответствует предметной области — в строительном магазине не смешивают обычно уже смешанные краски.

А выявить и решить проблему нам на самом деле помогло явное выписывание предусловий и инвариантов. Хорошо их выписывать не просто текстом, а в коде, с использованием операторов (или функций) `assert`, которые в том или ином виде присутствуют в любом нормальном языке. Это с одной стороны дополнительный текст, объясняющий, что происходит в модели, с другой стороны, работающий механизм, который поможет сразу выявить проблемы. Чем больше `assert`-ов — тем лучше.

# Domain-Driven Design, стратегические аспекты

Юрий Литвинов

yurii.litvinov@gmail.com

## 1. Целостность модели

Центральной идеей методологии предметно-ориентированного проектирования является единая модель предметной области и единый язык, что хорошо и правильно, но если над проектом работает сотня человек, это проблематично. Собрать всех на одной кухне, чтобы они могли оттачивать единый язык, физически невозможно, особенно если они на самом деле говорят на разных языках и сидят на разных континентах. В этой лекции речь пойдёт о том, как применять предметно-ориентированное проектирование к проектам, разрабатывающимся несколькими командами, о том, как вообще разрабатывать большие проекты, какие архитектурные проблемы при этом появляются и как их можно решать. Эта лекция по сути является кратким пересказом части IV книги «Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем», Э. Эванс, и заканчивает рассказ о книге.

Итак, предметно-ориентированное проектирование в случае проекта, над которым работают несколько команд, сталкивается с проблемой того, что команды неизбежно имеют разные видения продукта. Эту проблему можно решать,

- либо поддерживая модель интегрированной — но тогда затраты на поддержание её целостности будут слишком велики (в идеале — все команды должны будут непосредственно общаться со всеми остальными, и каждое изменение в модели утверждаться остальными командами), при этом модель наверняка получится слишком общей, чтобы быть полезной;
- либо приняв ситуацию как должное и разрешив модели быть фрагментированной — но тогда это неизбежно затруднит переиспользование кода в рамках проекта и интеграцию системы.

При этом бесконтрольное существование модели может привести к ошибкам, связанным с разным пониманием командами нюансов сущностей модели. Эрик Эванс приводил в книжке хороший пример, когда над системой работало несколько команд, и одной из них потребовалась абстракция для платежа клиента. Оказалось, что в коде уже был класс «Платёж», разработанный другой командой, ну и, понятно, его решили переиспользовать. Парочки полей не хватало, одно называлось не совсем так, как надо, но ладно, в конце концов, какое имеет значение конкретное название. Однако система через пару дней начала внезапно падать, в модуле оплаты счетов субподрядчику, для которого изначально был разработан класс «Платёж», в частности, на генерации налоговых отчётов. Стали разбираться, обнаружили в системе странные платежи, которые никто не вводил и которые

не имели никакого смысла. Оказалось, что система кренилась из-за того, что у странных платежей не было заполнено поле «необлагаемый процент», несмотря на то, что система требовала его заполненности и сама подставляла туда значение по умолчанию при создании платежа. Выяснилось, что это действительно были платежи клиентов, и для них действительно поле «необлагаемый процент» не имело смысла, поэтому просто не заполнялось. И эти платежи тоже попадали в вычисление налоговой отчётности (потому что они платежи же!) и всё падало.

Это выглядит как какая-то частная проблема, но корень зла тут вполне системный — две команды использовали одну сущность в разных, хоть и похожих, смыслах. И не имели никакого механизма, позволявшего выявить заранее несоответствие значений, которые команды вкладывали в термин «Платёж». Решением проблемы стало создание двух отдельных абстракций «Платёж поставщику» и «Платёж клиента» и договорённость более не мешать друг другу.

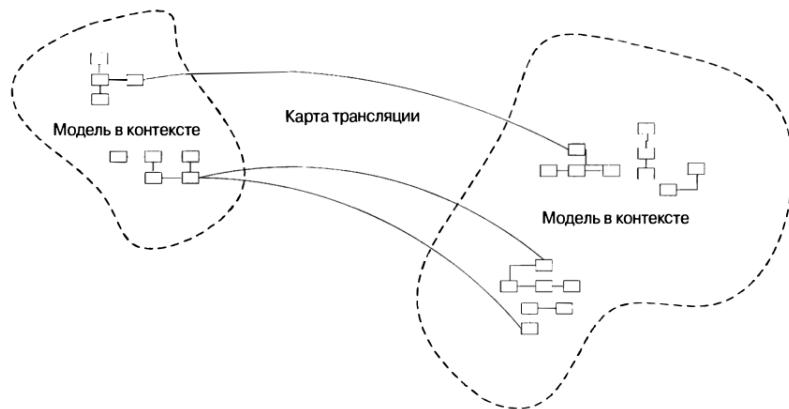
## 1.1. Ограниченный контекст

Командам из примера про «Платёж» могла бы помочь заранее достигнутая договорённость о границах их предметных областей и правилах переиспользования кода. В предметно-ориентированном проектировании для этого вводится понятие *«Ограниченный контекст»*. Ограниченный контекст (Bounded context) — это кусок предметной области (и, соответственно, реализующего её кода), в которой применима единая модель предметной области. Всё, что внутри ограниченного контекста, должно следовать этой единой модели, иметь единый язык и т.п., всё, что вне — не должно делать никаких предположений о модели и не имеет право её без спроса переиспользовать. То есть, ограниченный контекст — это что-то вроде атомарной области проекта, над которой работает одна команда (как клетка в живом организме, простите за банальную метафору). Разделение системы на ограниченные контексты обычно следует организационной структуре проекта, которая, в свою очередь, чаще всего следует высокуюровневой структуре системы. Например, в одном из проектов, в котором работал автор, по созданию средства автоматизированного реинжиниринга, была группа синтаксического анализа, группа извлечения бизнес-правил, группа генерации кода. Каждая группа имела собственный ограниченный контекст, свою терминологию и своё видение задачи, и интегрировалась с другими с помощью вполне определённых интерфейсов. Что происходило внутри, было делом каждой команды — например, группа извлечения бизнес-правил занималась вообще какими-то магическими алгоритмами статического анализа программ, которые никто, кроме них, не понимал.

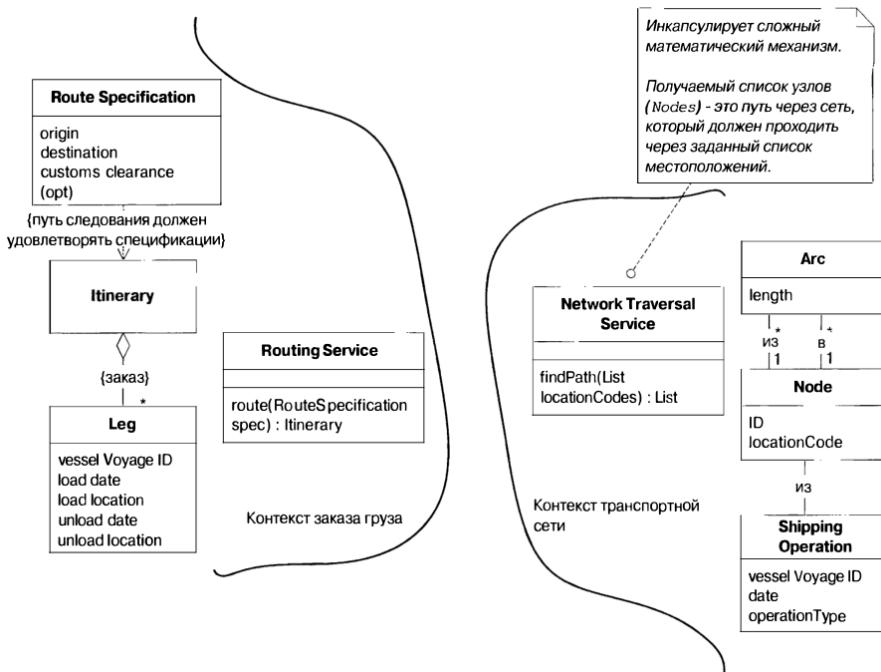
Внутри одного ограниченного контекста может работать довольно большое количество людей, между которыми также может возникнуть недопонимание, напряжение при попытке поддержания единого видения и т.п. Практика показывает, что 3-4 человека, тесно работая вместе, обычно могут договориться, но делить систему на ограниченные контексты по 3-4 человека оказывается непрактично. Поэтому в рамках ограниченного контекста практикуют *«непрерывную интеграцию»* в её классическом понимании — слияние изменений в основную ветку раз в несколько часов, постоянную (после каждого коммита) сборку и запуск юнит-тестов, обеспечение высокого тестового покрытия, непрерывное общение в рамках команды. Всё это позволяет быстро понять наличие проблем в понимании модели внутри ограниченного контекста и устраниить их.

Для интеграции с другими ограниченными контекстами могут использоваться «карты

«контекстов» (Context map). Карта контекстов фиксирует, как понятие из одной модели в рамках одного ограниченного контекста транслируется в понятие из другой модели из другого контекста. Карты трансляции обычно просто описываются на естественном языке (с применением терминов единых языков интегрируемых моделей), но могут и использоваться диаграммы такого примерно вида:



Вот любимый Эриком Эвансом пример про сервис грузоперевозок, показывающий взаимосвязь ограниченных контекстов в рамках одной задачи:



Есть команда, занимающаяся бизнес-логикой сервиса прокладки маршрута. Она опирается понятиями «Спецификация маршрута», «Маршрут», «Перевозка» и использует

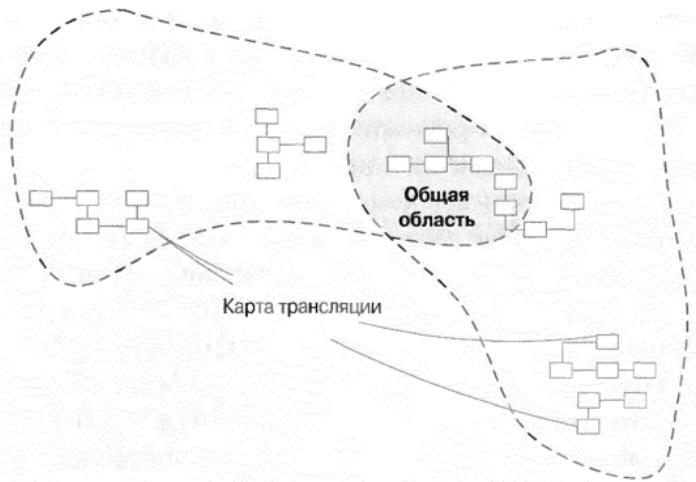
«Сервис прокладки маршрута» для вызова кода второй команды, который собственно составляет маршрут перевозки между двумя точками. Вторая команда работает в терминах алгоритмов на графах, она ничего не знает и не хочет знать про перевозки, спецификации маршрута и конкретные корабли или поезда, везущие грузы. У них есть узлы и дуги с весами, узлы привязаны к координатам на местности и в узлах можно осуществлять погрузочно-разгрузочные операции над некоторыми абстрактными транспортными средствами, идентифицируемыми своим кодом. Это позволяет второй команде переиспользовать существующие библиотечные реализации графов и алгоритмов на них, не мучаясь со спецификой предметной области. Но это, естественно, требует определённых усилий на интеграцию, поскольку «Сервис прокладки маршрута» со стороны первой команды должен выполнять трансляцию в термины «Сервиса поиска пути» второй команды и обратно, и о том, как работают эти сервисы, команды должны чётко договориться. Зато всё, что находится за этими сервисами, может разрабатываться независимо. Что, конечно, приводит к дубликации кода, потому что Leg и Arc, по сути, выражают одно и то же понятие, но в данном случае это абсолютно осмысленно — команды могут работать параллельно, не мешать друг другу и не мешать разные понятия в одну кучу.

## 1.2. Паттерны интеграции контекстов

Есть некоторые типичные ситуации, в которых возможны разные степени интеграции моделей предметной области и разные способы действий, необходимых, чтобы этой интеграции достичь и при этом не навредить разработке. Эванс в своей книге выделяет несколько подходов и даёт им имена (поэтому их можно рассматривать как что-то вроде паттернов интеграции контекстов). Выбор подхода, который стоит применять, зависит от конкретной ситуации и обстоятельств, скорее всего, не зависящих от команды. Поэтому стремиться к высокой интеграции контекстов не всегда стоит, и один из официальных подходов к интеграции — это отказаться от интеграции вообще.

### 1.2.1. Общее ядро

Первый рассматриваемый здесь паттерн интеграции — «Общее ядро» (Shared Kernel) — предполагает наиболее тесную интеграцию, кроме объединения двух ограниченных контекстов в один. Такой подход предполагает выделение общей части двух ограниченных контекстов и совместную реализацию элементов модели оттуда:



Классы из общей области должны иметь строго одинаковый смысл во всех интегрируемых ограниченных контекстах. Обычно это либо какое-то общее ядро, определяющие базовые понятия всей предметной области проекта, либо это могут быть какие-то классы с данными, которые используются для того, чтобы два ограниченных контекста могли легко и без дополнительных преобразований обмениваться информацией. При этом наличие общего ядра не отрицает применения карты трансляции к элементам модели, которые в общее ядро не попали.

Паттерн «Общее ядро» применим только в случае, если две команды могут непосредственно общаться друг с другом, находятся на одном уровне подчинения и не будут мешать друг другу. Классы из общего ядра можно менять только после согласования с обеими командами, так что это дело хлопотное, а необходимость избегать неправильного понимания сущностей и «ересей» в едином языке требует и постоянного неформального общения членов команд.

### 1.2.2. Заказчик-поставщик

Паттерн «Заказчик-поставщик» (Customer-Supplier) применяется, когда общение между командами затруднено и/или когда команды находятся в подчинённом отношении друг к другу (не обязательно административно подчинённом, это может быть подчинение в смысле зависимостей между компонентами — например, группа генерации кода может оказаться зависимой от группы синтаксического анализа, особенно если синтаксический анализ нужен не только генерации кода). Особенно паттерн актуален, когда команды имеют потенциально разный цикл релизов и ориентируются на разных конечных пользователей. «Общее ядро» в такой ситуации применять невозможно, поскольку попытка согласовать общее ядро может провалиться из-за близости релиза одной из команд, конфликтующих приоритетов или конфликтующих миропониманий. Тогда одна из команд рискует оказаться заблокированной.

Чтобы этого избежать, следует явно зафиксировать отношения между командами. Одна из команд выступает в роли заказчика (одного из заказчиков) — участвует в митингах по планированию, поставляет задачи. Вторая команда выступает в роли поставщика, стараясь

удовлетворить выставляемым требованиям, в соответствии со своим планом, приоритетами и видением предметной области. При этом очень желательно, чтобы команда-зачазчик предоставляла автоматизированные приёмочные тесты, чтобы обеспечить непрерывную интеграцию (в смысле, в котором этот термин используется в DDD) и уменьшить боль при приёмке. При этом обе команды могут работать в разных ограниченных контекстах, взаимодействующих через чётко определённые точки интеграции.

Паттерн предполагает, что обе команды находятся в одной иерархии управления, чтобы возможные конфликты не надо было эскалировать через всё руководство организации. Конфликты неизбежны, если у поставщика несколько заказчиков, а поскольку конфликты погут привести к полной блокировке команды-заказчика, они должны быстро и эффективно разрешаться.

### 1.2.3. Конформист

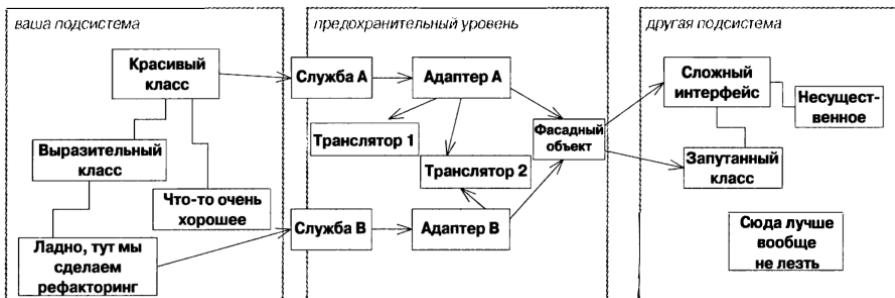
Паттерн «Конформист» (Conformist) применяется, когда команда полностью зависит от компонента, на который никак не может повлиять. Например, это может быть какой-то legacy-код или целое legacy-приложение, либо платформа, на которой мы ведём разработку, либо большая open-source-библиотека (в этих двух случаях как-то повлиять мы можем, большинство адекватных разработчиков принимают пуллреквесты или багрепорты, но это обычно хлопотно, небыстро и требует существенно большего погружения в код компонента, чем вам, возможно, хочется). Это может быть и технология, навязанная «сверху».

«Конформист» предполагает, что вы просто принимаете модель и миропонимание «основного» компонента, и подстраиваете свою модель предметной области под модель, которая там используется. Это обычно очень не нравится программистам из-за синдрома «not invented here» — часто всё, что сделали не лично мы, кажется нам каким-то бредом (тем более если это навязано сверху). Тем не менее, часто паттерн «Конформист» на самом деле хорошая идея, потому что legacy-код вполне может оказаться написанным командой, которая уже 20 лет в предметной области и знает о ней существенно больше, чем знаете вы, или имеете надежду узнать в разумные сроки. Практика показывает, что популярные сторонние компоненты обычно отражают вполне хорошее понимание предметной области, и принять его для своего кода правильно — может сэкономить массу времени на анализ и поможет избежать типичных ошибок.

### 1.2.4. Предохранительный уровень

Паттерн «Предохранительный уровень» (Anticorruption Layer) применяется, когда «Конформист» применить нельзя, потому что модель предметной области стороннего компонента вам противна (ну или просто не подходит), однако компонент своё дело делает. В этом случае рекомендуется реализовать программную прослойку между вашим кодом и компонентом, и использовать компонент через эту прослойку, чётко задокументировав, как понятия из одной модели преобразуются в другую. Обратите внимание, речь идёт не о механическом преобразовании данных в разные форматы (хотя это, безусловно, тоже может потребоваться), а именно о преобразовании понятий между разными ограниченными контекстами. То есть создание класса-обёртки просто потому, что вам название какого-то класса или метода не нравится — вполне оправдано. Неудивительно, ведь модель предметной области — это скорее про единый язык, чем про функциональность.

С технической точки зрения прослойка (тот самый «предохранительный уровень») может быть большим и сложным куском кода, выполняющим нетривиальные действия. При реализации могут помочь паттерны проектирования «Фасад», «Адаптер», службы (то есть статические классы), различного рода трансляторы, кеши и т.д.:



### 1.2.5. Отдельное существование

Паттерн «Отдельное существование» (Separate Ways) применяется, когда даже «Предохранительный уровень» неприменим (например, из-за дорогоизны разработки подсистемы преобразования), или вообще в ситуациях, когда преимущества от интеграции меньше затрат на неё. «Отдельное существование», как понятно из паттерна, предполагает отсутствие интеграции вообще — вы принимаете решения игнорировать существование другого проекта и не пытаться оттуда ничего переиспользовать.

Хороший жизненный пример применения такого паттерна — командная работа над дипломом. В некоторых ситуациях какая-либо запланированная интеграция реализаций разных дипломов может быть очень плохой идеей, даже если несколько дипломов пишутся в одном проекте по очень похожим задачам и могут активно переиспользовать функциональность друг друга. В этом случае риски и потенциальный ущерб от их осуществления существенно превышают возможные преимущества от интеграции, даже если сама интеграция и недорога — кто-то из участников может уйти в академ или просто не осилить, остальным тоже уходить в академ?

### 1.2.6. Служба с открытым протоколом

Паттерн «Служба с открытым протоколом» (Open Host Service) — это что-то вроде «Заказчик-Поставщик» со стороны поставщика, у которого уж очень много заказчиков, так что приглашать их всех на planning-сессии и учесть все их пожелания не получится. В этом случае рекомендуется в каком-то смысле вообще не рассматривать «хотелки» заказчиков (за исключением требований) и строить модель предметной области самим. После чего предоставить публичный интерфейс, представляющий эту модель, и опубликовать саму модель вместе с интерфейсом (обычно в виде текстовой документации). А заказчикам предлагается самим подстраиваться под опубликованную модель (применяя паттерн «Конформист», или «Предохранительный уровень», если они сильно недовольны).

Пример использования такого подхода — это большинство публичных REST-сервисов, таких как VK API, Google Drive API и т.д. Публикуются сами сервисы и документация, в которой объясняется, какими сущностями и как эти сервисы манипулируют. При

этом обратите внимание, что модель предметной области и единый язык опубликованного сервиса может отличаться от модели предметной области и единого языка, используемого в его реализации. Как правило, «внешнюю» модель стараются сделать как можно более простой, тогда как в реализации можно (и часто требуется) разгуляться по полной.

### 1.2.7. Общедоступный язык

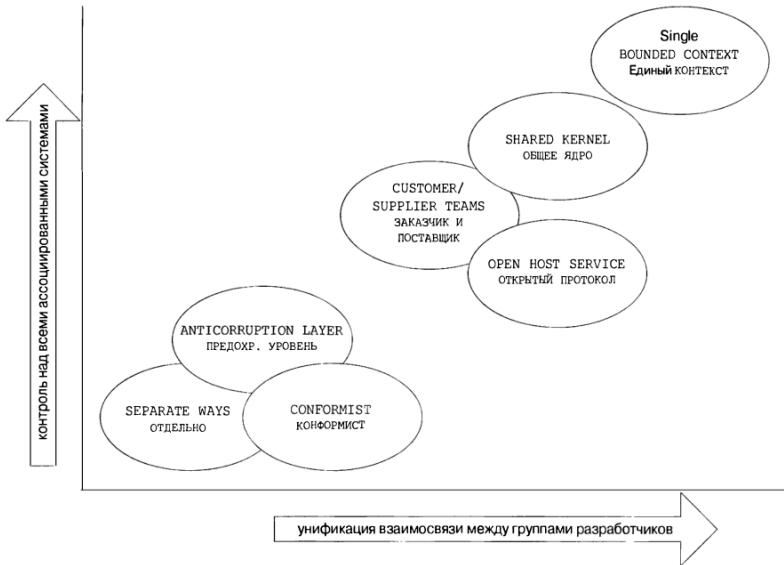
Паттерн «Общедоступный язык» (Published Language) применяется, когда предметная область достаточно популярна, чтобы в ней появилось очень много «заказчиков» и много «поставщиков». Тогда рекомендуется общими усилиями разработать модель предметной области, удовлетворяющую всех поставщиков, стандартизовать её и навязать заказчикам. Эта общая модель предметной области не должна навязывать ничего касательно реализации и должна быть максимально простой, но вместе с тем достаточно выразительной. Для реализации компонентов и сервисов в рамках этой модели могут использоваться свои ограниченные контексты (и не одни), не обязательно совпадающие с опубликованной моделью. Общая модель служит скорее языком и общей средой для общения различных систем в предметной области.

Примеры применения такого подхода — это языки программирования (например, компиляторы C++ поставляются несколькими известными производителями, но это более-менее один язык, то же с UML) и отраслевые стандарты — например, стандарты хранения и передачи медицинских данных, типовые архитектуры авионики или автомобильного бортового программного обеспечения. Стандарты фиксируют общую понятийную базу и терминологию (единий язык) и связь между понятиями (модель предметной области), которым должны следовать все, работающие по стандарту.

Кстати, стандарт IEEE 1016-2009, упоминавшийся в первой лекции этого курса, — это как раз пример стандарта с общедоступным языком. Там прямо с использованием UML описывалась предметная область архитектурных видов и предлагался единый язык.

### 1.2.8. Сравнение подходов к интеграции

Взаимосвязь различных подходов к интеграции ограниченных контекстов описывает рисунок из книги Эванса:



В правом верхнем углу находится полная интеграция — когда команды работают в едином ограниченном контексте. Это требует максимальной взаимосвязи между группами разработчиков (на самом деле, по факту это должна быть одна команда) и максимального контроля над кодом (на самом деле, кодовая база должна быть полностью подвластна команде). Чуть слабее требования у паттерна «Общее ядро». Дальше стоят «Заказчик-Поставщик» и «Открытый протокол» — при этом «Заказчик-Поставщик» позволяет влиять на компонент, от которого мы зависим, «Открытый протокол» в меньшей степени, зато требует существенно больше документации (поэтому требуется больше коммуникации, хоть и односторонней).

В левом нижнем углу находятся паттерны, применяемые, кода всё плохо. «Предохранительный уровень» — кода мы хотим писать какой-то код, чтобы защитить своё миропонимание от навязанного третьесторонним компонентом (так что требует много кода), «Конформист» — когда не хотим, но тогда нам надо больше разбираться в миропонимании компонента (оттого больше коммуникации, даже если она заключается в чтении документации и кода). И наконец, «Отдельное существование» не требует ни коммуникации, ни какого-либо контроля, но и интеграции никакой не даёт.

### 1.3. Пример: унификация слона

Небольшой метафорический, но на самом деле очень жизненный пример применения паттернов интеграции из книги Эванса основывается на стихотворении «Слепцы и слон» Дж. Г. Сакса (1816–1887):

Шесть седовласых мудрецов  
Сошли из разных стран.  
К несчастью, каждый был незряч,  
Зато умом блистал.  
Они исследовать слона

Явились в Индостан.

Один погладил бок слона.  
Довольный тем сполна,  
Сказал он: “Истина теперь  
Как божий день видна:  
Предмет, что мы зовем слоном,  
Отвесная стена!”

А третий хобот в руки взял  
И закричал: “Друзья!  
Гораздо проще наш вопрос,  
Уверен в этом я!  
Сей слон — живое существо,  
А именно змея!”

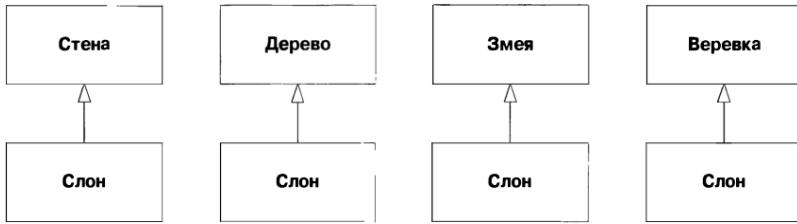
Мудрец четвертый обхватил  
Одну из ног слона  
И важно молвил: “Это ствол,  
Картина мне ясна!  
Слон — дерево, что зацветет,  
Когда придет весна!”

Тем временем шестой из них  
Добрался до хвоста.  
И рассмеялся от того,  
Как истина проста.  
“Ваш слон — веревка. Если ж нет  
Зашейте мне уста!”

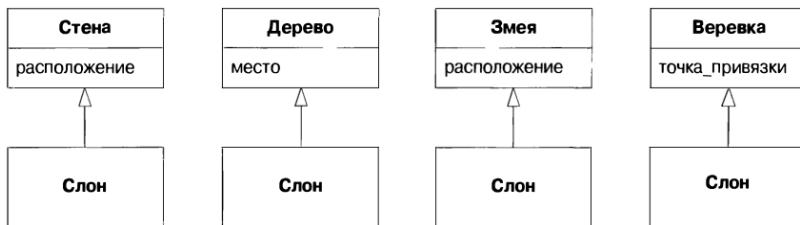
А как известно, мудрецам  
Присущ упрямый нрав.  
Спор развязав, они дошли  
Едва ль не до расправ.  
Но правды ни один не знал,  
Хотя был в чем-то прав.

Шесть мудрецов — это шесть команд разработчиков, работающих над одним проектом в шести ограниченных контекстах. Посмотрим, как они могли бы постепенно интегрировать свои контексты, улучшая в процессе интеграции качество модели и своё понимание предметной области.

Первый вариант интеграции — отсутствие интеграции как таковой, почти как в оригинальном стихотворении. Это соответствует паттерну «Раздельное существование», когда каждый мудрец имеет свою точку зрения, придерживается её и не лезет в другие ограниченные контексты. Системы не могут интегрироваться друг с другом и их модели предметной области радикально различны:



Некоторая минимальная интеграция могла бы быть получена с помощью паттерна «Предохранительный уровень». Каждый мудрец всё ещё считает, что остальные понимают предметную область неправильно, но они могут договориться о схеме трансляции между понятиями:

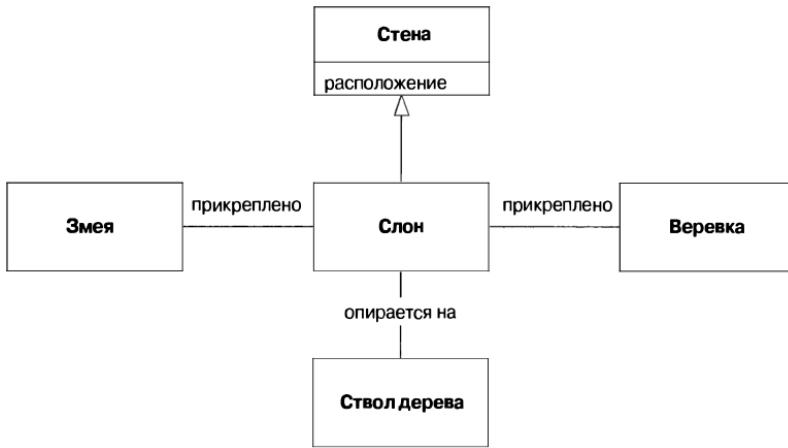


**Трансляция:** {Стена.расположение ↔ Дерево.место ↔ Змея.расположение ↔ Веревка.точка\_привязки}

Теперь у всех моделей есть некоторые свойства, которые по оговоренным заранее правилам могут преобразовываться друг в друга. Не всегда это преобразование тривиально, не всегда сохраняет все данные, не всегда имеет смысл в рамках «настоящей» предметной области, но всё-таки позволяет системам работать совместно. Например, мудрец, считающий слона стеной, может определить расположение стены, которое может быть так или иначе транслировано в точку привязки верёвки для мудреца, который считает слона верёвкой.

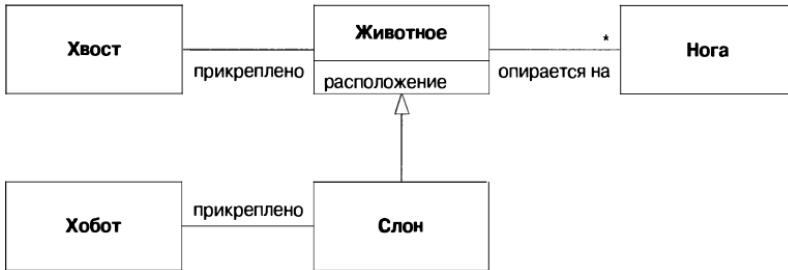
Стоит отметить, что и в реальной жизни «настоящая» предметная область далеко не очевидна, особенно в начале разработки, особенно если разработка ведётся в сфере, далёкой от жизненного опыта разработчиков (например, информационная система для автомобильного завода или даже информационная система поддержки учебных практик СПбГУ). Если несколько команд работают над одной областью и анализируют её независимо (например потому, что относятся к разным организациям и просто не знают о существовании друг друга), они будут действовать в точности как слепые мудрецы и проблемы на первых этапах интеграции, если она начнётся, практически неизбежны.

Следующий этап — это выделение общего ядра. В оригинальном стихотворении мудрецы просто разругались, в мире программирования всё же более типична ситуация, когда команды продолжают работать вместе и находят общие фрагменты своих моделей, что становится основой интеграции:



Тут мудрецы уже поняли, что говорят по сути об одном, но каждый всё ещё наделяет слона своими свойствами, специфичными для его точки зрения. Тем не менее, они уже могут рассуждать в терминах общей и понимаемой ими однозначно сущности «Слон». При этом одному из мудрецов повезло и его класс «Стена» был принят как предок для класса «Слон», так что мудрецы согласились, что у слона есть расположение. Но это не значит, что класс «Стена» сам принадлежит смысловому ядру: всё, что от него можно унаследовать, то есть состояние, поведение и инварианты — да, безусловно, но вряд ли мудрецы бы согласились включить в единый язык общего ядра тот факт, что слон является стеной. А в DDD общий язык важнее технических соображений.

Ну и последний этап интеграции — когда мудрецы достигают полного согласия и формируют единый ограниченный контекст:



Таким образом, команды разработчиков, в отличие от мудрецов, с помощью постепенной интеграции всё-таки смогли установить истину и собрать из фрагментов единую модель предметной области, в которой могут работать, пока снова не разругаются.

## 2. Смысловое ядро

Ключевой момент предметно-ориентированного проектирования больших систем — это выделение их самой содержательной части, вынесение из неё всего, непосредственно к

ней не относящегося, и аккуратный её дизайн. Эта содержательная часть предметной области называется *смысловое ядро* (Core Domain) — это то, что, собственно, делает систему ценной. То, за что, собственно, заказчики платят деньги. Например, в любой информационной системе есть библиотека для работы с базой данных, скорее всего, какая-то сетевая часть, скорее всего, какой-то UI. Но полезна информационная система только тогда, когда правильно моделирует данные и бизнес-процессы заказчика. Или компьютерная игра — вы можете потратить кучу времени на реализацию сохранения-загрузки, но купят игру либо из-за интересной игровой механики, либо из-за хорошего дизайна, либо ещё из-за чего-то, с сохранением и загрузкой не связанного. Сохранение и загрузка тоже важны, и без них игру не купят, но это не главное.

Только смысловое ядро фактически составляет know-how, то, что выделяет ваш продукт на фоне конкурентов и то, что представляет настоящую ценность, всё остальное в принципе можно смело выложить в open source (а лучше взять из open source, потому что наверняка уже десять раз реализовано). Пожалуй смысловое ядро должно быть самой проработанной частью системы. Причём, ирония состоит в том, что опытные программисты не любят заниматься смысловым ядром, потому что оно требует глубокого погружения в предметную область. Гораздо приятнее заниматься базами данных или UI и написать потом в резюме длинный список технологий, которыми ты мастерски владеешь, чем написать в резюме, что ты прекрасно разбираешься в работе логистических компаний (и тогда тебя трудоустроят те две-три компании в мире, которые делают свой бизнес на автоматизации в сфере логистики) или ты эксперт в размножении тушканчиков и умеешь строить отличные предметно-ориентированные модели.

С таким мировоззрением надо бороться. На самом деле, работать на резюме в плане изучения стеков технологий может быть полезно, пока вы junior developer, но ближе к позиции senior уже стратегически невыгодно. Богатый технический бэкграунд к этому моменту постепенно обесценится из-за естественной смены модных технологий, и с вами на рынке труда будут конкурировать junior-ы, которые на новых технологиях выросли, и поэтому знают их лучше вашего. Да и вряд ли вы к тому времени будете хотеть juniorскую зарплату. А вот специализация в некоторой предметной области juniorам недоступна, поскольку требует массы времени и усилий уже после базового образования, и, хоть и сужает возможности трудоустройства, делает вас ценным специалистом, которого самого приглашают на работу, причём на очень высокую зарплату (потому что такие специалисты редки). В мире миллионы Java-программистов, но программист, шарящий в размножении тушканчиков, возможно, один, и если кому-то он понадобится, за него отдаут любые деньги. Например, молодых выпускников матмеха, успевших специализироваться в компьютерной криминалистике, несмотря на небольшое количество работодателей в этой сфере, перекупали друг у друга у меня на глазах.

## 2.1. Дистилляция

Смысловое ядро, поскольку оно самая важная часть системы, должно быть как можно меньше (чтобы с ним легче было работать) и чётко отделённым от остальных компонентов системы. Процесс выделения смыслового ядра Эванс называет *дистилляцией* и приводит несколько дельных советов про то, как это делать.

Первый приём, позволяющий выделить смысловое ядро — это Domain Vision Statement. Domain Vision Statement — это короткий документ (примерно на одну стра-

ницу), описывающий смысловое ядро и его полезность. Обычно его имеет смысл делать в самом начале проекта, вместе с описанием проекта вообще. В реальной жизни ни разу такого не видел, однако видел “устав проекта”, где, в частности, и описывается то, что должно быть в Domain Vision Statement. Тем не менее, охотно поверю в полезность такого документа. Смыл тут в том, чтобы документ был коротким и ёмким, не надо описывать архитектуру ядра, надо описать, что оно умеет и зачем оно.

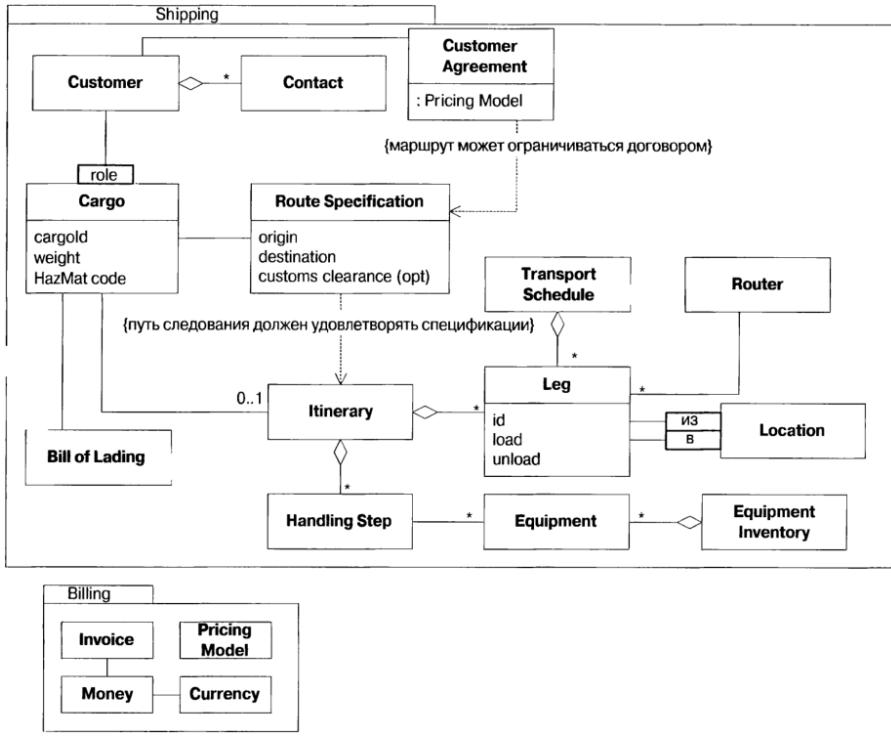
Выделенное ядро (Highlighted Core) — это способ уже на готовой модели обозначить смысловое ядро. Бывает двух видов. Первый — *дистилляционный документ* — это 3-7 страниц текста про то, что составляет смысловое ядро и как его элементы взаимодействуют друг с другом. По сути, развёрнутый Domain Vision Statement, с архитектурой смыслового ядра, как правило, часть архитектурной документации. Второй — Flagged Core — это когда элементы ядра выделены на существующей модели, например, специальным стереотипом или даже просто цветом. Дёшево и вместе с тем достаточно эффективно.

Далее выполняется собственно дистилляция, то есть избавление кода смыслового ядра от всего лишнего. Делается это вынесением неспецифичной для приложения функциональности в отдельные модули. Собственно видов неспецифичной функциональности в DDD выделяют два:

- Неспециализированные подобласти (Generic Subdomains) — куски кода, важные для предметной области, но неспецифичные для конкретной системы;
- Связный механизм (Cohesive Mechanism) — куски кода, неспецифичные для предметной области вообще. Это могут быть разные технические вещи, например, работа с графами. Иногда оказывается, что хитрая структура самописных классов — это не более чем замаскированный граф, и тогда стоит просто изменить архитектуру так, чтобы можно было переиспользовать одну из готовых библиотек. Связные механизмы принципиально отличаются от неспециализированных подобластей тем, что они уж точно тысячу раз реализованы и имеет смысл выбрать готовый компонент. С неспециализированными подобластями тоже может повезти, а может, их реализации можно взять из других проектов компании, но с большой вероятностью их придётся писать самим (хоть они и не смысловое ядро системы).

## 2.2. Пример, грузоперевозки

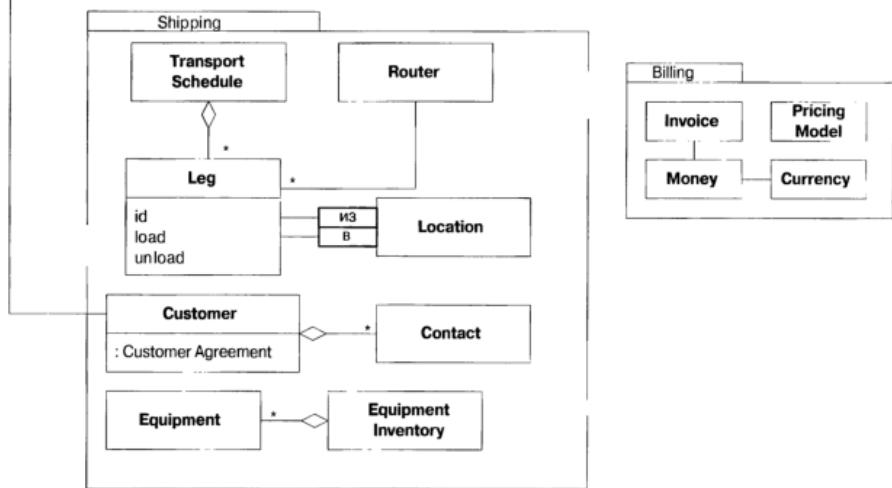
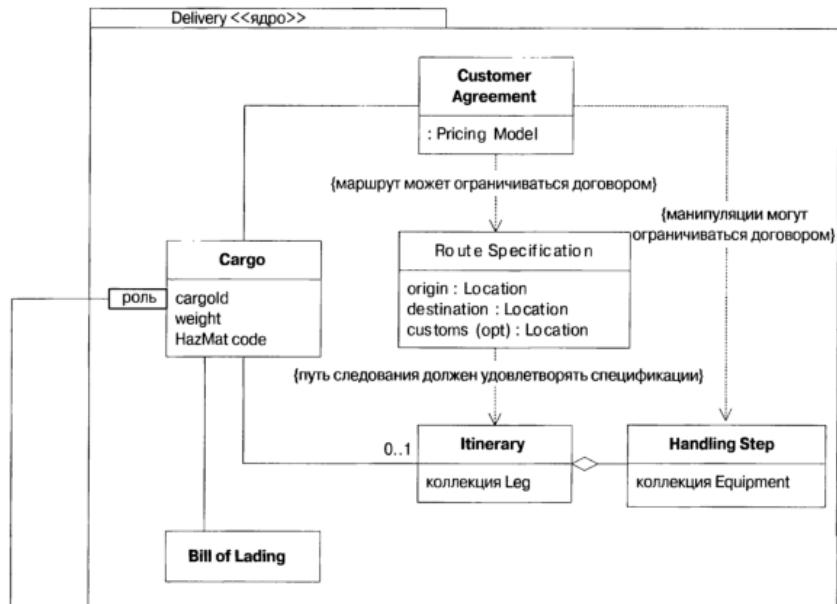
Рассмотрим дистилляцию модели на примере логистической системы с прошлой пары. Напомним, что мы проектировали систему, позволявшую заказать доставку груза (Cargo), по маршруту (Itinerary), удовлетворяющему спецификации (Route Specification) и состоящему из шагов (Handling Step) и перевозок (Leg):



Посмотрев на эту систему внимательно, мы понимаем, что тут есть классы, относящиеся собственно к доставке груза, к прокладке маршрута, к работе с клиентом и всякие явно вспомогательные классы. Поскольку основной бизнес нашей компании — это организация перевозок, смысловым ядром для нашей системы будут классы, которые непосредственно занимаются доставкой. Это Cargo (груз, мы его, собственно, доставляем), RouteSpecification (это то, откуда, куда и как мы везём груз, критическая для нашего бизнеса часть модели), Itinerary, Handling Step (это наша основная работа на самом деле), BillOfLading (ключевой для нашей работы документ) и CustomerAgreement (потому что всё это на самом деле нужно для выполнения обязательств по договору).

А вот сам Customer и его контакты к смысловому ядру не относятся — это умеют любые CRM-системы в разы лучше нас. Перевозка, расписание транспорта, даже прокладка маршрута — это не ядро нашего бизнеса, хотя и, безусловно, важно. Большая часть этого — это вообще задачи транспортных компаний, и соответствующие классы мы используем просто как данные. Прокладка маршрута и составление расписания интереснее, но есть специальные библиотеки, которые этим занимаются и умеют делать хорошо — опять-таки, это важная часть нашего бизнеса, но не то, за что нас будут любить клиенты, не то, что мы хотим уметь делать лучше всех в мире.

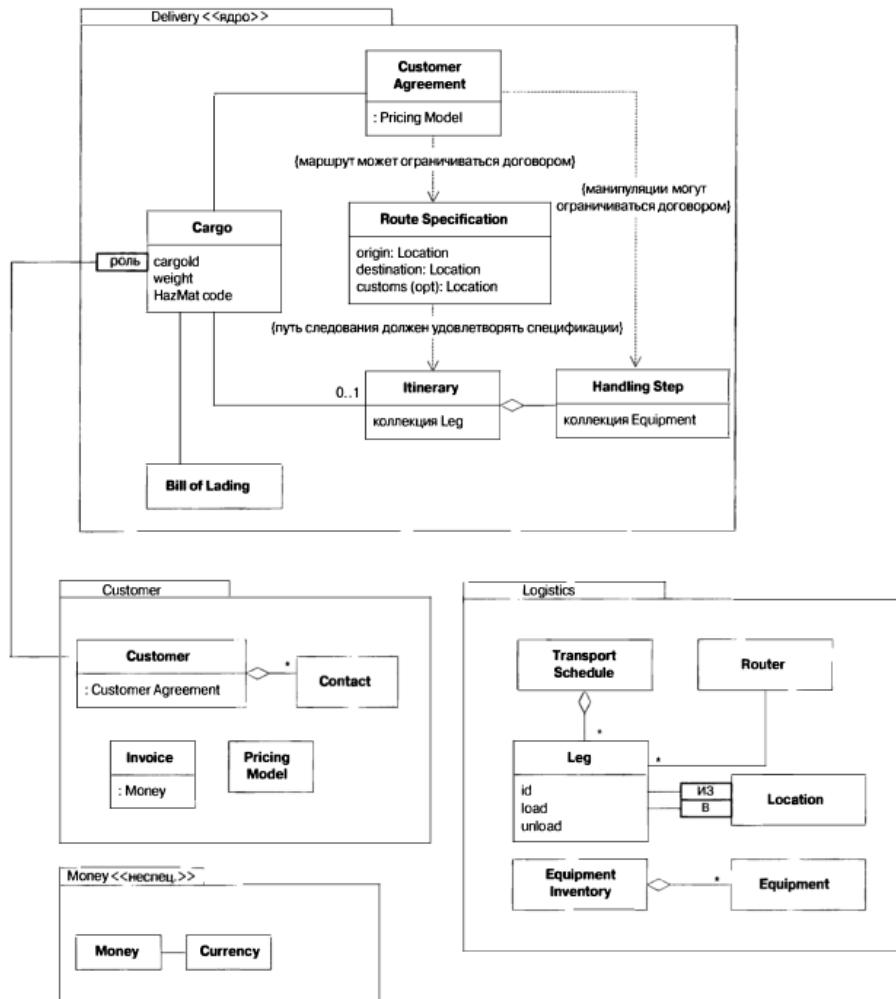
В результате рефакторинга может получиться вот такая модель («flagged core»):



Как видим, сложность смыслового ядра уменьшилась вдвое.

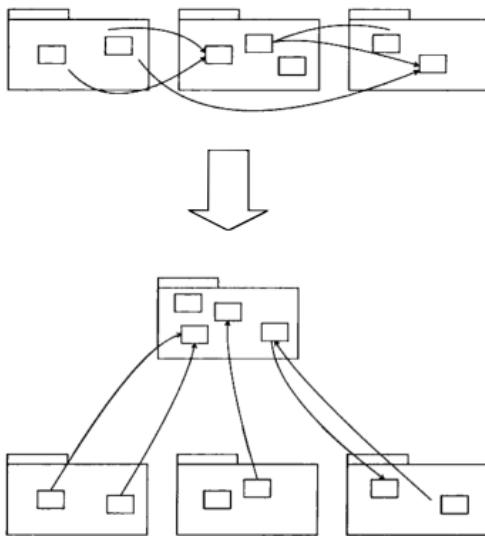
Однако и на этом останавливаться ещё рано — модуль Shipping нарушает принцип единственности ответственности, так что его можно разбить на два: Customer и Logistics. В Customer пойдут классы, ответственные за работу с клиентом (Customer, Contact), в Logistics — всё, что связано с перевозками. При этом мы можем заметить, что в модуле Billing есть классы Invoice и PricingModel, которые специфичны для нашего бизнеса, и Money с Currency, которые нет. Давайте вынесем Invoice и PricingModel в модуль Customer (это оправданно потому что модель ценообразования и выставление счетов явно относятся к клиенту и вполне могут от него сильно зависеть), и тогда можно Money с Currency оставить.

вить в одном модуле и сделать его неспециализированной подобластью. Тогда получится вот такое:



## 2.3. Абстрактное ядро

Иногда даже после этих ухищрений смысловое ядро остаётся всё ещё слишком большим, чтобы его можно было адекватно сопровождать. Тогда можно применить приём «Абстрактное ядро» — когда смысловое ядро состоит только из абстрактных классов/интерфейсов, которые потом реализуются отдельными модулями. Радикально сложность модели это всё равно не уменьшит (но так и должно быть, это «essential complexity»), но хотя бы позволит избавиться от ненужных подробностей реализации. Схема такая:



### 3. Крупномасштабная структура

Крупномасштабная структура — это самая «архитектурная» часть архитектуры, набор основных правил, которым следует вся система или группа систем. Например, может быть задекларировано, что система в целом имеет уровневую архитектуру, и уровни такие-то и такие-то. Тогда как реализация самих уровней может выполняться в совершенно разных архитектурных стилях. Более того, просто задекларировать, что система имеет уровневую архитектуру — это уже зафиксировать её крупномасштабную структуру, хоть и слишком общую, чтобы быть реально полезной. Вообще, крупномасштабная структура системы — это чаще всего уточнённый под конкретную систему архитектурный стиль плюс набор специфичных для системы ограничений (например, перечисление основных компонентов и их ответственостей, правила, по которым в систему можно добавлять новые компоненты).

Крупномасштабная структура не фиксируется раз и навсегда, как бы ни был велик соблазн это сделать. По мере роста понимания предметной области в процессе *переработки знаний* и по мере получения обратной связи о том, насколько удобна выбранная архитектура в коде крупномасштабная архитектура вполне может меняться и эволюционировать (естественно, совместно с моделью и с кодом). Кроме того, не следует делать её слишком жёсткой и пытаться всё предусмотреть (раз она может эволюционировать, её всегда можно будет потом поправить). Тем более не следует использовать административный ресурс, чтобы принуждать разработчиков ей следовать. Организация команды «Архитектор в башне из слоновой кости», когда архитектор разрабатывает архитектуру, но сам код не пишет и не прислушивается к обратной связи от программистов, зарекомендовала себя на практике очень плохо, кроме того, в предметно-ориентированном проектировании её особенно не любят (поскольку она ещё и осложняет ту самую переработку знаний).

Небольшие проекты вполне могут жить без явно задокументированной крупномасштабной структуры — если у вас всего пять классов, то их взаимоотношения будут вполне

достаточны в качестве архитектуры. Более того, пытаться «архитектурить» небольшой скрипт или утилиту может быть саботажем сроков и стоимости проекта, особенно если реализация архитектурных мечтаний требует написания большого количества кода.

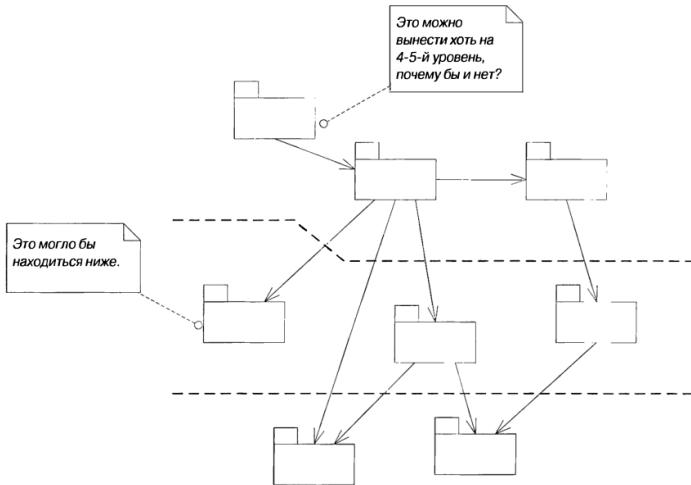
Но надо принимать во внимание, что все большие проекты начинались как небольшие, и если о крупномасштабной структуре не подумать, в какой-то момент возникнет хаос. Причём, совершенно не требуется сразу, как только вы написали первый маленький классик, продумывать плагинные системы, миграции, шины интеграции и т.п., ведь крупномасштабная структура может эволюционировать. Сначала стоит зафиксировать хоть какую-то, затем постепенно её корректировать и уточнять.

Ещё стоит отметить, что крупномасштабная структура — это не огромный архитектурный документ с изобилием UML-диаграмм и ссылок на стандарты (хотя если такой есть, то это не плохо), это вполне может быть устная традиция, передаваемая от разработчика к разработчику. Вообще, самый полезный носитель крупномасштабной структуры по DDD — это *единий язык* и его термины. Если в ходе общения сам собой выкристаллизовался слой А, слой Б и появился набор правил в духе «нет-нет, мы так не делаем» — это уже вполне нормально в качестве крупномасштабной структуры, особенно если все понимают её однозначно (а должны, это ведь единый язык).

В этом плане очень полезна *метафора системы* — некая аналогия, определяющая, как в целом понимать систему. Разные метафоры используются в программировании по-всеместно, из-за незримости ПО и желания всё-таки его как-то представлять. Например, метафора рабочего стола для пользовательских интерфейсов, файервола в сетевой безопасности и т.д., даже деревья в программировании называются деревьями, потому что похожи на... деревья (а могли бы — связными ациклическими графами). Однако метафору не всегда легко подобрать, и она не всегда оказывается удачной.

Особо опасна метафора тем, что тащит за собой лишний смысл из реальной жизни. Например, файервол ассоциируется с противопожарной перегородкой, разделяющей разные части сети, и если понимать её слишком буквально, можно забыть, что файервол должен контролировать и локальный трафик. Или тот же рабочий стол — в реальной жизни не бывает рабочих столов, бесконечных во все стороны, поэтому и рабочие столы на компьютере ограничены (хотя, наверное, было бы удобно иметь скроллящийся как карта в реал-тайм стратегиях рабочий стол, особенно управляемый движением головы, взглядом или вообще VR). Тем не менее, хорошая метафора, особенно из предметной области проекта, может сильно помочь.

Наиболее типичная крупномасштабная структура — это уроневая. Уровневый архитектурный стиль мы неоднократно уже обсуждали, но есть ещё одно важное соображение, касающееся использования уровней как именно крупномасштабной структуры — нельзя чисто механически делить систему на уровня топологической сортировкой графа зависимостей компонентов. Например,



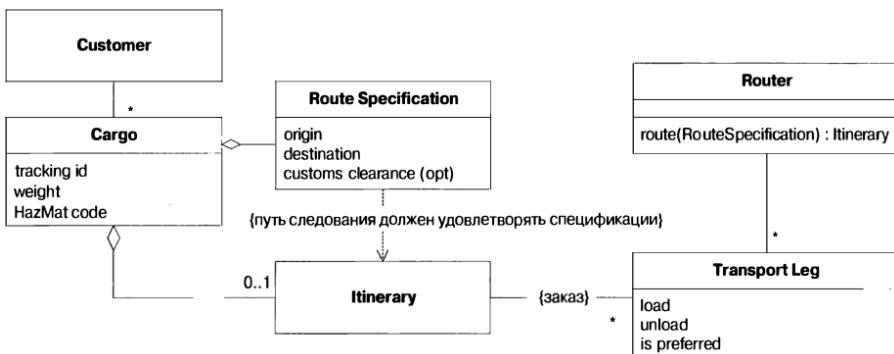
Разбиение по уровням, как и любое разбиение в архитектуре вообще, должно сообщать дополнительную информацию об идеи реализации системы, выражать мысль архитектора и объяснять происходящее. Причём, объяснять — на самом деле, самое важное: качество архитектуры, как научной теории в представлении философов науки 20-го века, определяется её объясняющей способностью. Не важно, что архитектура позволяет быстро или эффективно что-то реализовать, она всё равно плоха, если не объясняет происходящего.

Кстати, поэтому результаты автоматической генерации диаграмм по коду — не архитектура ни разу.

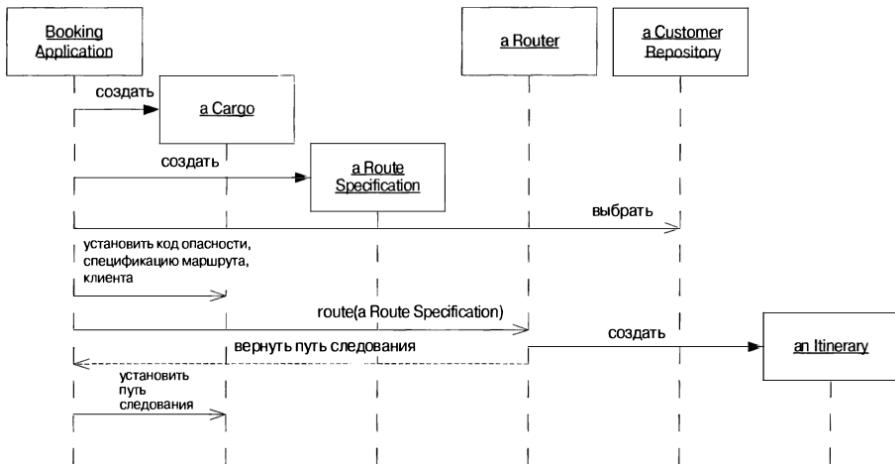
Чтобы избежать бессмысленного разбиения на уровни, надо каждому уровню придать какое-то значение из предметной области и ограничения, с ним связанные. Рассмотрим эту мысль подробнее на примере.

### 3.1. Пример, перевозка грузов

Пример рефакторинга системы к осмысленной уровневой крупномасштабной структуре, как обычно, рассмотрим на системе про грузоперевозки. Напомним, что у нас есть груз, который надо доставить в соответствии со спецификацией маршрута, для чего мы с помощью класса Router генерим расписание (Itinerary) перевозки:



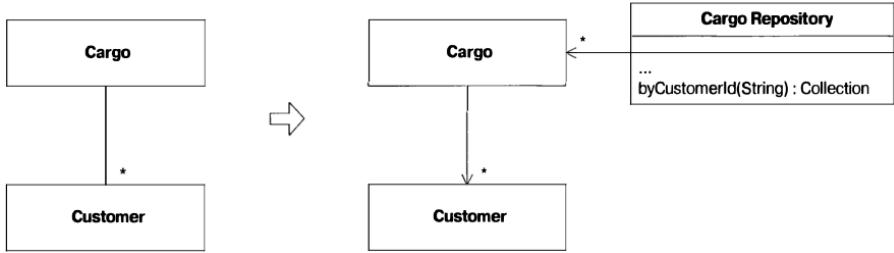
Система небольшая, так что кажется, что искусственно бить её на уровни нет смысла. Но мы понимаем, что система будет расти, и хотим придумать какое-то простое правило, которое бы объясняло, куда добавлять какие классы и как они должны вписываться в уже существующую систему, чтобы не получился огромный клубок из взаимосвязанных классов. Для начала рассмотрим поближе, как в текущей архитектуре работает типичный запрос, заказ перевозки:



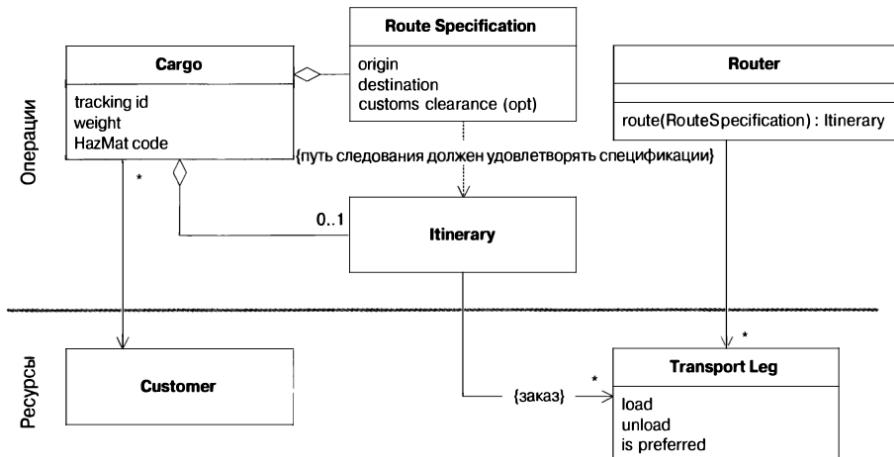
Приложение, бронирующее перевозку, сначала создаёт Груз, коорый надо перевести, затем Спецификацию маршрута (где Груз забрать и куда и когда его надо доставить). Далее оно выбирает из репозитория объект Клиент, который использует для инициализации Груза, и отдаёт то, что получилось, в Маршрутизатор, который создаёт и возвращает Расписание перевозки. Расписание устанавливается Грузу, и после этого он готов к поездке.

Можно заметить, что планирование маршрута использует клиента просто как данные, и строит расписание, которое на самом деле состоит из Перевозок, которые реализуются компаниями-перевозчиками (в отличие от нас, владеющими реальными средствами перевозки), так что для нас Перевозки — это тоже просто данные. Возникает идея это явно отразить в архитектуре, разделив её на два уровня — Операционный и Ресурсный. Ресурсный уровень представляет то, что обеспечивает наши возможности (перевозчики, и, как ни странно, клиент — он нам деньги платит, заказывает перевозки, и мы им не управляем никак). Операционный уровень — это то, как мы пользуемся нашими возможностями, тут будут классы, отвечающие за оперативное управление процессами и планирование. Очевидно, Customer и Transport Leg пойдут на ресурсный уровень, остальное — на операционный.

Ещё мы хотим топологическое ограничение — объекты уровня ниже не могут знать про объекты уровня выше (иначе толку от разделения на уровни никакого). Однако в нашей исходной модели Customer знает про Cargo (ну а что, он же должен знать про заказанные перевозки), и про Transport Leg не указаны направленности ассоциаций. Во втором случае проблем нет, перевозчик ничего не должен знать про Маршрутизатор и Расписание даже по логике вещей, а вот с Customer и Cargo реально потребуется рефакторинг. Но мы уже умеем так делать с прошлой лекции: используем паттерн «Репозиторий»:



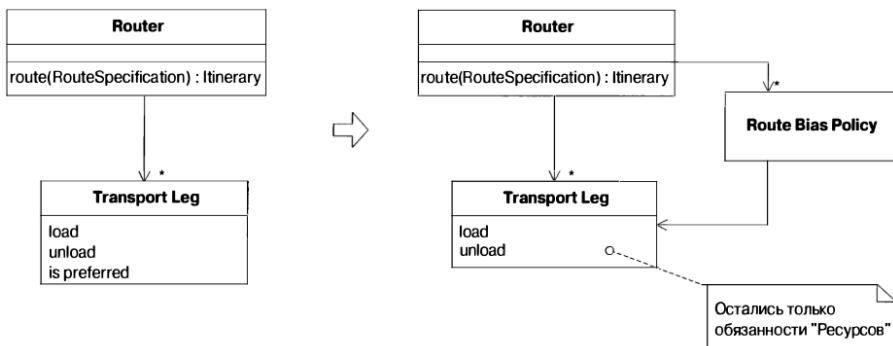
Теперь Груз знает про Клиента, но Клиент не знает про груз, однако может при желании получить его из репозитория грузов. Все репозитории глобально доступны, так что в уровневую структуру в любом случае не вписываются, они как бы живут параллельно. Топологии системы это не ломает, потому что репозитории ссылки не хранят, да и на них никто не хранит ссылок. После рефакторинга получилось как-то так:



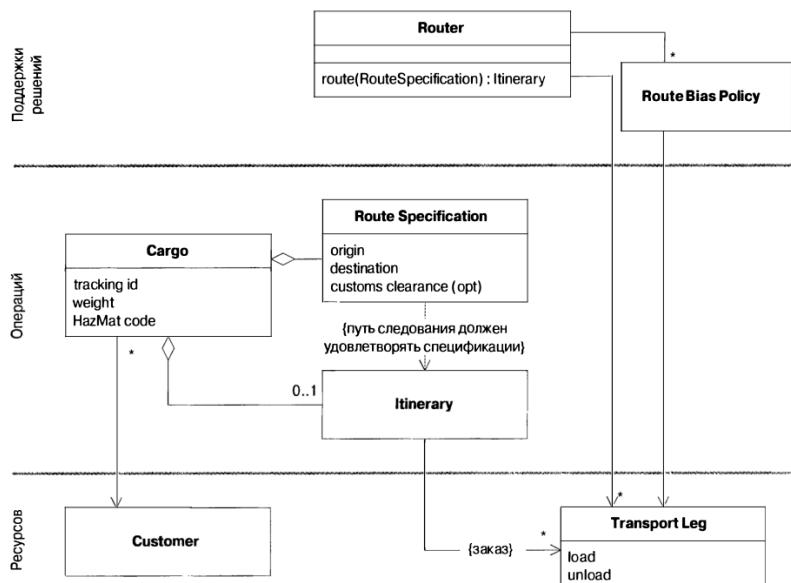
Стало лучше. Однако вспомним, как мы определили операционный уровень: «это то, как мы пользуемся нашими возможностями, тут будут классы, отвечающие за оперативное управление процессами и планирование». «управление процессами и планирование» звучит так, будто на самом деле мы хотим два уровня, «управление процессами» и «планирование» — уровень Поддержки решений будет работать только иногда и строить план, который должен претворять в жизнь Операционный уровень. Очевидно, на уровень Поддержки решений отправляется класс Router, который по данной Спецификации маршрута должен построить нам Расписание. Спецификацию маршрута и Расписание можно оставить на операционном уровне, потому что про них должен знать Груз, чтобы исполнять нужные перевозки (на самом деле, насчёт Спецификации маршрута спорно, но при заказе мы указываем груз и Спецификацию маршрута, и они всегда ходят вместе в нашей системе, так что давайте пока так — это типа маршрутного листа, который едет вместе с грузом, ему тогда самое место на операционном уровне).

Есть, однако, проблема, проискающая из предметной области. У Transport Leg есть поле «is preferred», которое true, если перевозку мы выполняем на нашем же транспортном средстве или силами компании-партнёра. Такие перевозки для нас предпочтительны,

потому что зачем нам делиться прибылью с чужими людьми? Однако это явно поле, нужное только на этапе планирования, и нужное только для Маршрутизатора при построении Расписания. Поэтому давайте его поднимем на уровень Поддержки решений, заведя отдельный класс Route Bias Policy, который бы помнил, какие из Перевозок нам больше нравятся. На самом деле, это позволит нам потом реализовать более мощные механизмы маршрутизации, чем даже были задуманы изначально (например, Route Bias Policy может при планировании маршрутов давать меньший вес Перевозкам вокруг Сомали или следить за новостями и не отправлять груз самолётом над зоной активного военного конфликта). Получилось что-то такое:

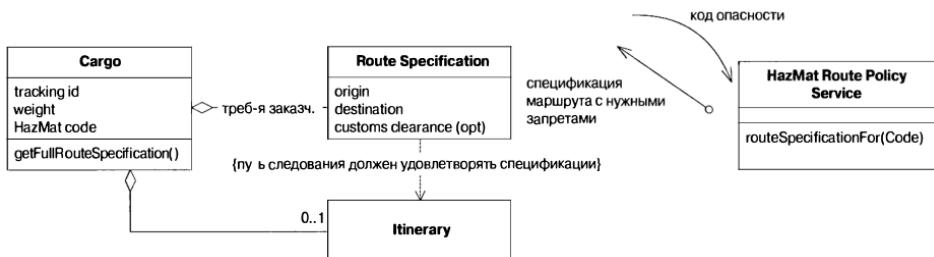


Ну и итоговая архитектура уже с тремя уровнями:

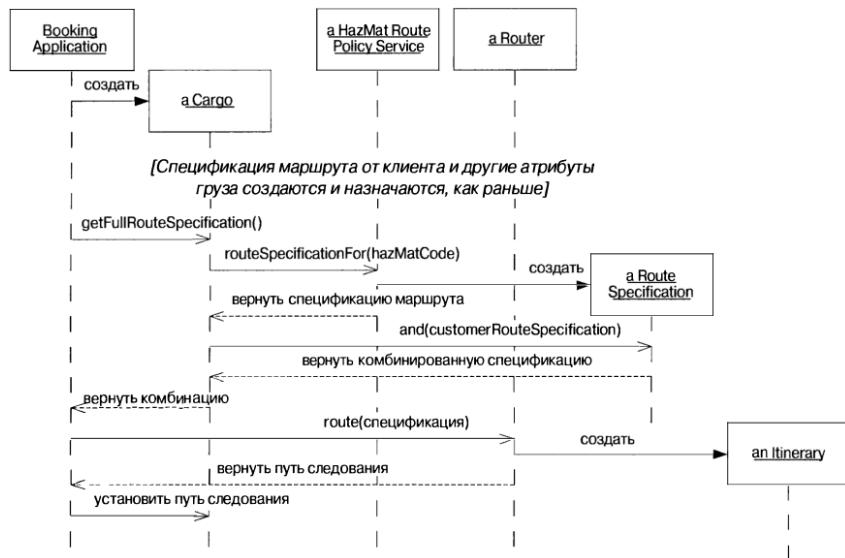


Стало ещё лучше, каждый уровень достаточно небольшой, есть чёткое понимание (ну, более-менее чёткое), на каком уровне что должно находиться, есть строгие топологические ограничения насчёт направленности ассоциаций.

И тут внезапно появляется новое требование — работа с опасными грузами. Их нельзя ввозить в определённые локации (было бы плохой идеей везти радиоактивные отходы через крупный город) и нельзя везти определёнными перевозчиками (попытки доставить Чужих на Землю для изучения, заразив команду корабля или колонистов, постоянно оканчивались провалом). Для поддержки таких дел Грузу добавляется поле «*HazMat code*», которое, хоть и нужно только на этапе составления Расписания, является неотъемлемым свойством груза, и выносить его в отдельную политику идеологически не очень хорошее решение. Поэтому первое решение, что пришло нам в голову — это сделать сервис, который бы брал груз и Спецификацию маршрута, и дополнял бы её запретами для конкретного кода опасности:



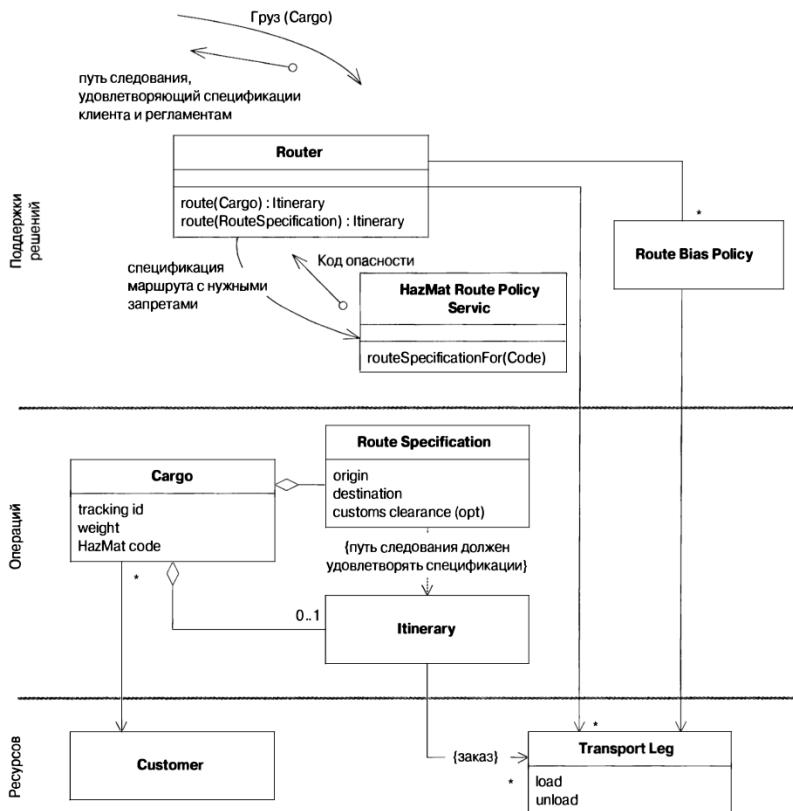
Тогда наша система работала бы так:



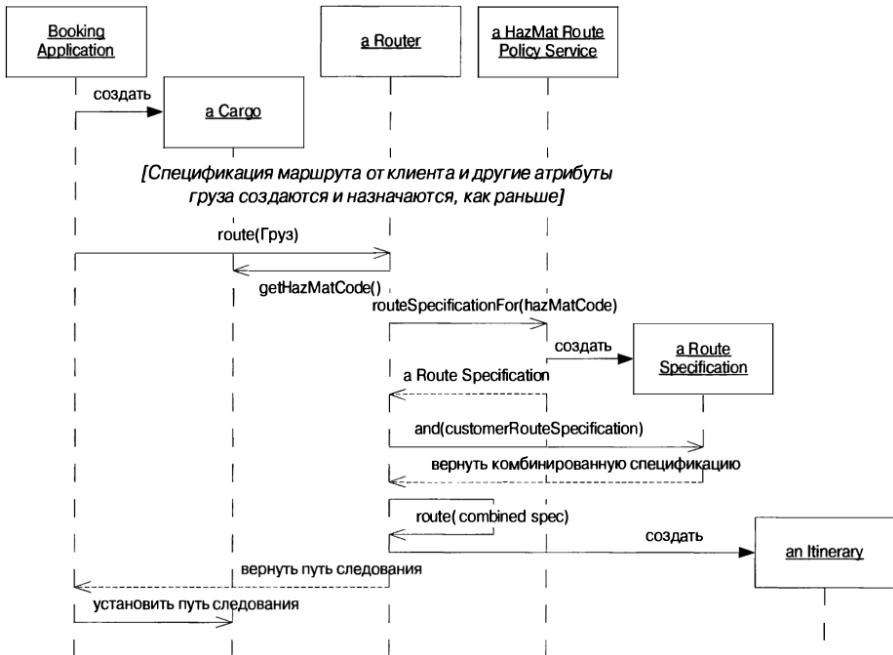
Приложение, в котором Клиент заказывает доставку, как обычно создавало бы груз и Спецификацию маршрута, после чего вызывало бы у Груза получение обновлённой спецификации, для чего Груз запрашивал спецификацию ограничений для своего кода опасности и мержит её (с помощью метода `and`) с обычной Спецификацией маршрута, которую нам заказал Клиент (паттерн «Спецификация» с композитной спецификацией таки). Ну а

далыше Маршрутизатор по данной комбинированной Спецификации маршрута составляет Расписание, которое и назначается Грузу.

Всё хорошо, но нарушает наше ограничение — политика работы с опасными грузами относится к уровню Поддержки решений, но её вызывает Груз, что заставляет либо перенести её на уровень ниже (что против идеологического разделения на уровни), либо позволить Грузу делать запрос к классу уровня выше (что против топологических ограничений, и тут даже трюк с репозиторием не поможет, поскольку Грузу реально требуется что-то содержательное делать с этим классом). Поэтому предложенная нами архитектура не подойдёт, но её легко исправить: давайте не Груз будет получать для себя Спецификацию маршрута, а сам Маршрутизатор (и когда нас посещает эта идея, мы удивляемся, как нам вообще пришла в голову предыдущая архитектура, ведь такое решение очевидно лучше даже без всяких опасных грузов — *переработка знаний во всей красе*). Вот что получается:



И теперь бронирование перевозки выглядит так:



Груз теперь ничего про Спецификации маршрута знать не хочет, Маршрутозадача просто принимает Груз, спрашивает его код опасности, формирует по нему Спецификацию маршрута, мерджит её с заказанной Клиентом Спецификацией маршрута, создаёт Расписание и возвращает приложению. Если Клиент доволен, Расписание выставляется Грузу, и вперёд.

Мораль этой истории такова, что крупномасштабная структура имеет приоритет над сиюминутными требованиями, и если требуется рефакторинг для её поддержания, его однозначно стоит провести, вместо того, чтобы легко и быстро реализовать новую функциональность, слегка отклонившись от установленных ограничений. Потому что потом получится архитектурная эрозия, Lava Flow и смерть проекта от собственной сложности.

### 3.2. Типичные уровни

Такое разделение на уровни, которое у нас получилось в примере, на самом деле не случайно и часто встречается в реальных системах в разных вариациях. Например, для систем автоматизации производства типична следующая четырёхуровневая крупномасштабная структура:

Принятия решений	Аналитические механизмы	Практически отсутствует как состояние, так и его изменение	Анализ управления Оптимизация использования Сокращение рабочего цикла
Регламентный	Стратегии Связи-ограничения (на основании целей или закономерностей данной отрасли)	Медленное изменение состояния	Приоритет изделий Предписанные регламенты изготовления деталей ...
Операционный	Состояние, отражающее реальное положение дел (деятельности и планов)	Быстрое изменение состояния	Инвентарная опись Учет состояния незаконченных деталей ...
Потенциальный	Состояние, отражающее реальное положение дел (ресурсов)	Изменение состояния в среднем темпе	Возможности оборудования Наличие оборудования Перемещение по территории ...

↓  
Зависимость

Самый нижний уровень, Потенциальный, как и у нас, отражает наши возможности — оборудование, производственные площади, контракты с постоянными поставщиками и т.д. Операционный уровень отвечает за оперативное управление — это конкретно какие детали сейчас есть на складе, статус производства, оперативный план и его выполнение, рабочие на рабочих местах. Регламентный уровень отвечает за ограничения, которых мы должны придерживаться при управлении и планировании — технические регламенты, законодательство, локальные акты и требования (график отпусков, например), стратегия предприятия. Уровень Принятия решений отвечает за стратегическое управление — анализ и оптимизацию рабочего цикла.

Каждый уровень имеет разный темп изменения ситуации: потенциальный меняется относительно редко — оборудование или помещения вряд ли часто меняются в течение одного рабочего дня. Данные на операционном уровне обновляются постоянно. Данные на регламентном уровне обновляются вообще раз в несколько лет. Уровень принятия решений своего состояния как правило вообще не содержит, пользуясь данными, которые предоставляют уровни ниже. Это можно использовать для планирования схемы БД и оптимизации запросов.

Для финансовых систем более типична тоже четырёхуровневая структура, но с другими уровнями:

Приняты решения	Аналитические механизмы	Практически отсутствует как состояние, так и его изменение	Анализ рисков	
			Анализ портфелей	Средства ведения переговоров
Регламентный	Стратегии Связи-ограничения (на основании целей или закономерностей данной отрасли)	Медленное изменение состояния	Пределы резервов Цели размещения активов	...
Обязательство	Состояние, отражающее сделки и договоры с клиентами	Изменение состояния в среднем темпе	Соглашения с клиентами Соглашения по синдикации	...
Операционный	Состояние, отражающее реальное положение дел (деятельности и планов)	Быстрое изменение состояния	Состояние кредитов Начисления Выплаты и распределения	...

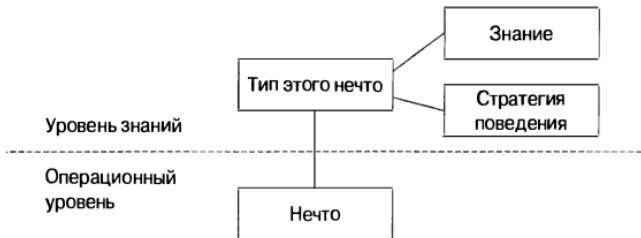
Зависимость ↓

Тут Потенциальный уровень отсутствует вообще, потому что финансовые системы не требуют особого оборудования (по крайней мере, того, чем надо реально управлять — если вы не Сбербанк), они оперируют деньгами. Собственно финансовые операции проводятся на операционном уровне, самом нижнем в такой схеме. Однако добавляется уровень Обязательств — это на самом деле то, в чём заинтересована финансовая организация и то, что она на самом деле продаёт (например, кредит — выглядит он так, будто вам продают деньги, но на самом деле это обмен обязательствами). На уровне Обязательств существуют договора, сделки и т.д. Регламентный уровень и уровень Принятия решений имеют тот же смысл и те же особенности, что и для производственных систем.

### 3.3. Другие высокоуровневые структуры

Помимо уровневой, бывают и другие высокоуровневые структуры, причём тысячи их. Например, Подключаемые компоненты (Pluggable Component Framework) — когда у нас есть ядро системы и плагинная система, которая позволяет динамически подключать/отключать плагины, которые и реализуют большую часть полезной функциональности. На самом деле, большинство современных систем в той или иной степени используют такой подход, кто-то целиком полагаясь на плагины (Visual Studio Code, в каком-то смысле IDEA), кто-то используя их очень ограниченно (например, браузеры — хотя это зависит от предпочтений пользователя). Даже Linux можно рассматривать как плагинную систему, где есть относительно простое ядро и куча кода, который в него встраивается с помощью чётко определённых интерфейсов.

Ещё интересная структура — это «Уровень знаний», когда система содержит в себе данные операционного уровня и метаданные (данные о данных) на уровне знаний:



Уровень знаний — это обычные объекты в памяти, просто они определяют, что и как можно делать с объектами операционного уровня (например, рефлексия — на уровне знаний объекты типа Type/Class, на операционном уровне — обычные объекты). Такой подход мы использовали, естественно, при проектировании систем визуального моделирования и метамоделирования, там правила визуального языка естественно представляются с помощью уровня знаний, который можно редактировать даже прямо во время выполнения, от чего операционный уровень начинает вести себя по-другому.

В финансовых системах такой подход позволяет на уровень знаний вынести финансовые продукты — например, хитрые правила начисления процентов по вкладам, которые постоянно меняются и реализовывать их в коде было бы весьма неразумно. Регламенты с уровня знаний там затем применяются к финансовым потокам на операционном уровне, при этом система становится очень гибкой — при наличии годного редактора правила может менять даже финансист, который программировать вообще не умеет. Конечно, у такого подхода есть и недостаток — логика уровня знаний не проверяется компилятором и имеет ту семантику, которую криворукие программисты реализуют, так что чрезмерное увлечение такими делами (когда у вас вся система представляется в виде правил на XML, например) может превратить поддержку в настоящий кошмар.

И последняя мысль по поводу крупномасштабной структуры — это то, что крупномасштабная структура для системы одна, но разные архитектурные стили не исключают друг друга, и, например, можно использовать уровневую структуру для системы в целом и «Уровень знаний» для реализации какого-нибудь из уровней, а плагины — для какого-то другого.

# Лекция 15: Примеры архитектур

## 1. Введение

В этой лекции мы рассмотрим примеры архитектур разных приложений из совершенно разных предметных областей, одно даже шуточное, но довольно поучительное. Начать имеет смысл с первоисточников, по которым подготовлена эта лекция:

- репозиторий Enterprise Fizz-Buzz: <https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition>;
- обзор архитектуры Bash в «The Architecture of Open Source Applications»<sup>1</sup>, правда про Bash из J. Garcia et al., «Obtaining Ground-Truth Software Architectures»<sup>2</sup>;
- краткий обзор архитектуры Git в той же «The Architecture of Open Source Applications»<sup>3</sup> и, что полезнее, но длиннее, глава 10 Git Book<sup>4</sup>;
- обзор архитектуры Battle for Wesnoth снова из «The Architecture of Open Source Applications»<sup>5</sup>.

## 2. Enterprise Fizz-Buzz

Начнём мы с самого серьёзного реального приложения — FizzBuzz Enterprise Edition, созданного серьёзными бизнесменами для серьёзных бизнес-задач. Проект, естественно, сатира, высмеивающая безумные практики enterprise-программирования в Java-мире, но нам будет полезна как пример овердизайна. Мне кажется, проще понять, что такая хорошая архитектура, если на одном конце шкалы будет один большой класс, который всё делает, а на другом — FizzBuzz Enterprise Edition с его подсистемой возврата перевода строки и прочими интересными архитектурными решениями. К тому же, там на самом деле используются (разумеется, не по делу) паттерны и архитектурные приёмы, которые были в этом курсе.

Собственно, серьёзная бизнес-задача такая:

Для чисел от 1 до 100:

- если число делится на 3, вывести «Fizz»;

<sup>1</sup> Глава про Bash в AOSA: <http://aosabook.org/en/bash.html> (дата обращения: 10.12.2020)

<sup>2</sup> Публикация на IEEE Xplore: <https://ieeexplore.ieee.org/document/6606639> (дата обращения: 10.12.2020)

<sup>3</sup> Глава про Git в AOSA: <http://aosabook.org/en/git.html> (дата обращения: 10.12.2020)

<sup>4</sup> Git Book: <https://git-scm.com/book> (дата обращения: 10.12.2020)

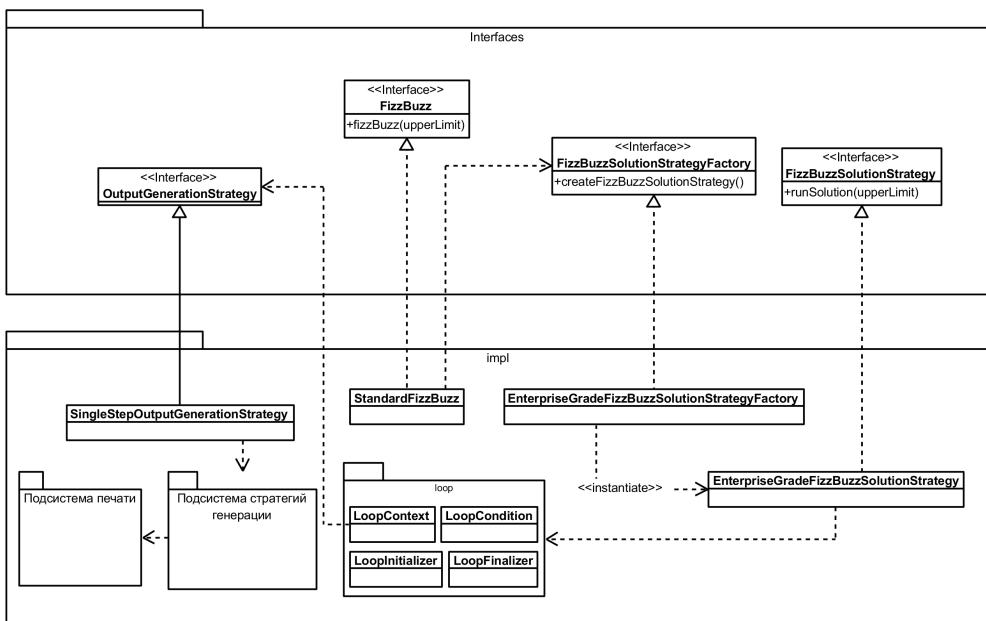
<sup>5</sup> Глава про Battle for Wesnoth в AOSA: <http://aosabook.org/en/wesnoth.html> (дата обращения: 10.12.2020)

- если число делится на 5, вывести «Buzz»;
- если число делится и на 3, и на 5, вывести «FizzBuzz»;
- во всех остальных случаях вывести само число.

«Правильное» архитектурное решение можно найти тут: <https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition>.

## 2.1. Обзор архитектуры

Вот обзорная диаграмма, описывающая в общих чертах структуру этого проекта:



По данным OpenHub.net, реализация на момент анализа проекта содержала 1662 содержательные строки кода и очень немного комментариев. OpenHub.net оценивает стоимость разработки этого проекта в 18K долларов.

Собственно, как всё работает: main создаёт ApplicationContext — класс из Spring Framework, который строит систему из доступных классов, используя принцип Dependency Injection. Общая идея Dependency Injection состоит в том, что процесс создания объектов, их сборки и инициализации сложной системы — отдельная большая подзадача, которую не стоит делать прямо в «боевом» коде, иначе нарушится принцип единственности ответственности. Правильнее автоматизировать этот процесс, описав набор интерфейсов, использовав эти интерфейсы в «боевых» классах, а затем доверив сторонней библиотеке (в нашем случае, Spring) пройтись рефлексией по нашей кодовой базе и каждому, кто в конструктор принимает объект некоторого интерфейса, автоматически подставить созданный объект класса, реализующего этот интерфейс.

Сама логика работы реализуется классом StandardFizzBuzz, который сам решать задачу не умеет, но умеет создать стратегию решения (через фабрику стратегий, потому что он не хочет знать о конкретном классе-стратегии). Стратегия создаёт подсистему управления циклом (ну раз в условии написано про числа от 1 до 100), подсистема состоит из классов, отвечающих за границы цикла, условие завершения, и главное, действия, выполняемые на каждом шаге цикла. Ещё есть хранилище текущего контекста вычисления цикла, хранящее счётчик цикла, такие дела.

Механизм вычисления одного шага цикла тоже использует паттерн «Стратегия» — сам он не хочет ничего считать, но делегирует вычисление стратегии генерации вывода. Стратегия генерации вывода принимает выходные данные, которые соответствуют счётчику цикла, но она не хочет знать про хранилище состояния цикла, в котором эти данные находятся, поэтому применяется адаптер (класс со звучным названием LoopContextStateRetrievalToSingleStepOutputGenerationStrategy), роль которого состоит в конвертации контекста цикла в параметр стратегии генерации. А вот стратегия генерации содержит в себе три стратегии вычисления, лежащие в одном списке, и для каждого значения счётчика цикла передаёт это значение всем этим трём стратегиям. Эти стратегии — FizzStrategy, BuzzStrategy и NoFizzNoBuzzStrategy, которые про параметр цикла говорят, надо с ним что-то делать или нет (так что на самом деле это паттерн «Спецификация», хотя в коде он так не называется — стратегии сами ничего не делают). Стратегии определяют, можно ли печатать Fizz, Buzz или FizzBuzz, после чего вызывается подсистема генерации, которая и печатает строку на экран.

## 2.2. Чему можно научиться

Положительные стороны этого решения такие.

- Separation of Concerns — один класс решает строго одну задачу, при этом для каждой задачи из предметной области можно указать класс, который ей занимается. Это хорошо, потому что если что-то изменится в условии, надо будет менять, скорее всего, вполне конкретный класс, результаты изменения будет легко проверить юнит-тестами (ну, по идеи, в этом проекте с юнит-тестами беда). Это же способствует переиспользованию кода и вообще сопровождаемости программы — из маленьких кирпичиков легко собрать новую программу в той же области, чем пытаться пилить под свои задачи большие куски кода.
- Dependency Inversion — реализация не должна зависеть от другой реализации, обе они должны зависеть от абстракции. В этом проекте почти никто не знает о других классах, есть чётко описанные интерфейсы, и класс может только вызывать методы по интерфейсу. Почему это хорошо — резко снижается связность. Не важно, как реализован тот или иной интерфейс — пока есть реализация всех нужных ему интерфейсов, наш класс будет работать. Мы можем дописывать новые реализации, изменять старые, пробовать разные реализации и даже выбирать их в зависимости от ситуации — при этом придётся переписывать минимум кода.
- Dependency Injection — становится возможным благодаря активному применению принципа Dependency Inversion. Мы можем динамически создавать и вставлять реализации интерфейсов в классы, которым эти реализации нужны. При этом использу-

зуется сторонняя библиотека Spring Framework, что тоже хорошая идея — неспецифичные для предметной области задачи, скорее всего, уже давно решены, и лучше, чем вы когда-либо сделаете.

- Паттерны:
  - «Фабрика» — для создания объектов, реализующих интерфейсы, позволяет легко менять реализацию и отделяет использование от создания;
  - «Стратегия» — инкапсулирует алгоритм работы, позволяет легко менять алгоритмы;
  - «Посетитель» — тут не особо нужен, но вообще позволяет отделить логику обхода структуры данных, состоящих из нескольких классов, от действий, выполняемых при обходе;
  - «Адаптер» — чтобы объект, реализующий один интерфейс, мог использоватьсь объектом, ожидающим другой интерфейс; опять-таки, уменьшает связность системы;
  - «Спецификация» — инкапсулирует сложное логическое условие или запрос; тут используется, чтобы определиться, когда что печатать;
  - «Цепочка ответственности» — позволяет сделать список (или вообще как-то организовать цепочку) объектов, каждый из которых может обработать запрос, либо отдать дальше, если не может; тут это не совсем «Цепочка ответственности», потому что запрос может быть обработан двумя стратегиями сразу.

## 2.3. Почему если бы это была домашка, она была бы не зачтена

- Не выполняется принцип Keep It Simple Stupid. Хорошее решение должно быть максимально простым из тех, что всё ещё решают задачу и способны внятно объяснить способ её решения. Чем сложнее решение, тем сложнее его понять и, соответственно, сопровождать. Закладывать в архитектуру что-либо «на будущее» надо с умом, потому что, как правило, будущее так и не наступает (или наступает, но другое), а за это надо платить сложностью
  - Кстати, какой-то достойный человек (жалко не помню кто) говорил «Неправильно говорить “строк кода написано”, правильно — “строк кода израсходовано” на решение той или иной задачи». Производительность труда программиста глупо мерять количеством строк кода, потому что из двух решений одной задачи лучше то, которое короче и проще. То есть лучше тот программист, который по традиционным метрикам работает хуже. Однако это не значит, что код лучше вообще не писать, или, тем более, писать всё в одну строку, используя только однобуквенные идентификаторы — *объясняющая способность программы* важнее её размера. Программа как книга — некоторые были бы лучше, если бы были короче (как EnterpriseFizzBuzz), некоторые были бы лучше, если бы были длиннее (как многие олимпиадные решения).
- «Синтаксическое» разделение на пакеты, а не «семантическое». Есть пакет interfaces, есть пакет adapters и т.д. Мне как читателю программы такое разбиение

на пакеты ничего не говорит, я и так вижу, что там, например, одни интерфейсы. А вот то, что классы, отвечающие за определение, Fizz это или Buzz, классы, отвечающие за печать, и класс, отвечающий за решение в целом, свалены в одну папку — не очень, потому что надо вчитываться в код, чтобы понять, как они взаимосвязаны. Выше была показана структурная диаграмма проекта, там были выделены подсистемы, и эти подсистемы, хоть и выделяются по смыслу (и выделяются в потоке управления), никак не выделяются в коде. На самом деле, это индикатор более глубокой и серьёзной проблемы — отсутствие модульности как таковой. Классы хотят и не взаимодействуют напрямую, но представляют собой единую массу, которая вместе решает задачу, а не набор блоков, каждый из которых решает свою подзадачу и понятным образом связан с другими блоками. Это проявление антипаттерна «Big Ball of Mud» — система не имеет крупномасштабной структуры, вся её архитектура определяется взаимодействием между конкретными классами.

- Хардкод основных параметров вычисления, иногда безумный:  
`public static final int INTEGER_DIVIDE_ZERO_VALUE = 0;`. Если уж enterprise, всякие штуки типа верхней и нижней границ цикла правильно грузить из XMLного конфига (ну или менее enterprise, JSON).
- Нет юнит-тестов, есть только несколько довольно жалких интеграционных тестов. Я бы не стал рефакторить такую систему. Вообще, возможности для юнит-тестирования тут очень высоки — чёткое разделение ответственостей позволяет писать изолированные тесты, следование принципу Dependency Inversion позволяет вовсю применять мок-объекты.
- Вообще нет логирования — если система упала, то непонятно, где и почему. Для enterprise-приложений логирование обязательно (и реально может спасти если не жизнь, то карьеру-то точно).
- 1663 строки кода и всего 40 строк комментариев. Архитектурного описания нет вообще (а на самом деле, очень бы пригодилось). Вновь пришедшему в проект человеку надо будет прочитать практически весь код, чтобы понять, как оно работает. Проблема особенно усугубляется отсутствием модульности и крупномасштабной структуры.

### 3. Архитектура bash

Теперь перейдём к более «настоящему» проекту — командному интерпретатору Bash. Это не тот пример хороший пример хорошей объектно-ориентированной архитектуры (он был написан на C, причём ещё в те времена, когда объектно-ориентированная архитектура была не очень модна), к тому же его архитектура не очень подробно документирована. Но нам этот проект интересен тем, что его часто (ну, относительно) используют в исследованиях по архитектуре как «подопытного кролика». Суммарно Bash имеет около 70К содергательных строк кода, к тому же ещё использует библиотеку Readline, которая, хоть и поддерживается тем же товарищем, что и Bash, и разрабатывалась в основном для Bash-а, но формально к нему не относится и не имеет кода, специфичного для Bash.

Дальнейшее изложение будет вестись по книге A. Brown, G. Wilson, The Architecture of Open Source Applications и статье J. Garcia et al., Obtaining Ground-Truth Software Architectures (рисунки ниже оттуда и из слайдов prof. N. Medvidovic). Книга, кстати, весьма годная, редакторы собрали довольно большое количество не очень больших заметок об архитектуре известных проектов с открытым исходным кодом от их авторов или maintainer-ов. С одной стороны, каждый из авторов описывает архитектуру как умеет, поэтому там всё очень неформально, не очень подробно и далеко не всегда соответствует правилам, про которые рассказывалось на этом курсе. С другой стороны, как редакторы справедливо отмечают во введении, архитекторы, которые проектируют здания, в ходе учёбы изучают сотни проектов зданий и критики этих проектов от профессионалов, тогда как архитекторы ПО, как правило, знакомы только с несколькими крупными системами, и то большинство из них они сами же и проектировали. Связано это с очень большой сложностью ПО и невозможностью зачастую восстановить архитектуру по программе, так что возможность посмотреть, как другие проектируют ПО, и поучиться на их ошибках или перенять их хорошие идеи может быть очень ценной.

Вот как выглядит диаграмма, характеризующая на высоком уровне архитектуру Bash:

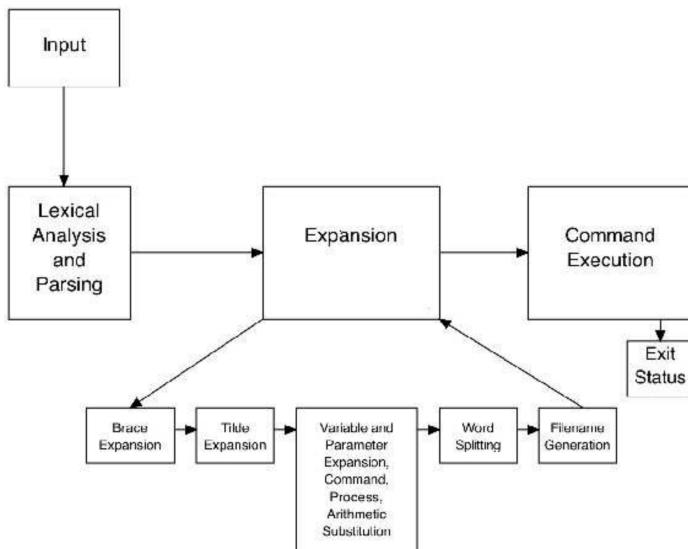


Диаграмма показывает поток данных между основными компонентами системы (в основном, Exit Status на самом деле компонентом системы не является, но я когда-то говорил, что неформальные диаграммы из квадратиков и стрелочек очень популярны и весьма полезны). Ввод (с консоли или из файла) поступает на вход лексическому и синтаксическому анализатору, далее последовательность слов, полученная парсером, отдаётся expansionу, который на самом деле представляет собой конвейер, последовательно применяющий преобразования к последовательности слов. Сначала выполняется подстановка фигурных скобок, затем тильды, затем доллара, затем разделение на слова (после парсера, такие дела), затем раскрытие шаблонов. Дальше то, что получилось, подаётся на вход исполнялке команды, которая отвечает за перенаправление ввода-вывода, пайпы, группы процессов и т.д., в итоге получается результат выполнения команды в виде кода возврата.

Всё общение между компонентами выполняется с помощью структуры WORD\_DESC и различных контейнеров, содержащих эти структуры. Вот определение этой структуры:

```
typedef struct word_desc {
    char *word; /* Zero terminated string. */
    int flags; /* Flags associated with this word. */
} WORD_DESC;
```

А вот так, например, представляются аргументы команды:

```
typedef struct word_list {
    struct word_list *next;
    WORD_DESC *word;
} WORD_LIST;
```

### 3.1. Ввод с консоли

За ввод с консоли отвечает библиотека Readline, которая отвечает за редактирование командной строки и за хранение истории команд. Она устроена как цикл «считать символ с клавиатуры — найти команду, ему соответствующую — исполнить её — показать результаты». Символ считывается как 8-битный символ (в те времена, когда это писалось, юникода ещё не было) и используется как индекс в «таблице диспетчеризации». В этой таблице может быть либо команда (например, «перейти в начало строки»), либо другая такая же таблица (это для поддержки сочетаний из нескольких символов), либо команда «вывести считанный символ». Ещё есть макросы (которые просто вставляют во входной поток символы). Все выводимые символы хранятся в буфере редактирования, а когда надо вывести результат на экран, Readline хитро рассчитывает минимальный набор команд управления курсором, который преобразует текущую отображаемую на экране строку в желаемую. Все внутренние данные хранятся в виде 8-битных символов, но Readline знает (теперь) про юникод и умеет его корректно отображать и корректно считывать позиции для многобайтовых символов.

Readline может быть расширена добавлением произвольных функций в таблицу диспетчеризации, и Bash этим пользуется, добавляя более 30 своих команд (например, автодополнение).

### 3.2. Синтаксический разбор

Readline возвращает просто строку, введённую пользователем. Первое, что с ней делается — лексический анализ, то есть, в случае с Bash-ем, разделение по словам и их идентификация. С последним дела обстоят довольно плохо, потому что смысл последовательности символов сильно зависит от контекста, так что лексер и парсер должны тесно общаться друг с другом, чтобы разбирать, например, вот такой ужас:

```
for for in for; do for=for; done; echo $for
```

Эта команда, кстати, вполне корректна и выведет на экран «for».

Bash — один из немногих шеллов, написанный на lex + bison, о чём, впрочем, автор несколько сожалеет, говоря, что ручная реализация рекурсивным спуском сильно упростила бы дело. Оригинальная грамматика шелла Борна, которую Bash пытается поддерживать, никому не известна, есть грамматика (контекстно-зависимая), стандартизованная POSIX, её-то и реализует Bash (так что грамматика Bash таки известна и документирована).

Интересно, что подстановка alias-ов выполняется лексером. Правда, для этого парсер сообщает ему, разрешена ли в данный момент подстановка. Лексер же отвечает за кавычки и бэкслеш.

Дополнительные проблемы создаёт подстановка результата выполнения команды и программируемое автодополнение, где тоже могут выполняться произвольные команды прямо в процессе разбора другой команды. Для поддержки таких вещей парсер умеет сохранять своё состояние и корректно восстанавливать его после разбора и исполнения «подкоманды».

Результат работы парсера — одна команда (которая в случае сложных команд типа for может содержать подкоманды), которая передаётся модулю, отвечающему за подстановки.

### 3.3. Подстановки

Подстановки (expansions) работают на уровне слов и могут порождать новые слова и списки слов. Они могут быть довольно сложными, например,

```
 ${parameter:-word}
```

раскрывается в *parameter*, если он установлен, и в *word*, если нет. А

```
 pre{one,two,three}post
```

раскрывается в

```
 preonepost pretwopost prethreepost
```

Ещё бывает подстановка тильды и арифметическая подстановка. Все они выполняются по порядку, то есть подстановщики организованы во что-то вроде конвейера.

Результат подстановки снова разбивается на слова (при этом это делает код, видимо, отличный от лексера). После разбиения происходит замена шаблонов — каждое слово интерпретируется как потенциальный шаблон и пытается сопоставиться с файлом в файловой системе.

### 3.4. Исполнение команд

Команды бывают встроенными (которые модифицируют состояние самого шелла, например, cd) и внешними, которые сами отдельные программы (например, cat). Ещё бывают сложные команды — if, for и т.д.

Каждая команда позволяет перенаправлять свой ввод и вывод (да, даже в for можно направить входной поток из пайпа). Самое сложное в реализации перенаправления — это следить за тем, когда его нужно отменить. Встроенные и внешние команды работают с точки зрения пользователя одинаково, поэтому наивная реализация перенаправления вывода

встроенной команды перенаправила бы вывод всего шелла. При этом Bash ещё и следит за файловыми дескрипторами, которые участвуют в редиректе, создавая новые или используя старые при необходимости.

Встроенные команды реализованы как синые функции, принимающие список слов как аргументы, и работают как «обычные» команды, только без запуска отдельного процесса. Единственная тонкость в том, что некоторые команды принимают как аргумент присваивания (например, export), они обрабатывают присваивание по-особому (для этого используются флаги в WORD\_DESC). Обычные присваивания (которые не в export или declare) реализованы на самом деле тоже как команды, но парсятся и обрабатываются немного по-особому. Если присваивание стоит перед обычной командой, то оно передаётся команде и действует до её завершения, если присваивание одно на строке, оно действует на весь шелл.

Внешние команды перед запуском ищутся в файловой системе, при этом шелл смотрит на PATH. Но если в имени команды есть слэши, то не ищет, а исполняет как есть. При этом единожды найдённая команда запоминается в хеш-таблице и дальше сначала ищется в ней. Если команда не нашлась, Bash вызывает функцию, которую можно переопределить, и многие дистрибутивы это делают, предлагая поставить нужный пакет с командой.

Ещё есть некоторые хитрости с управлением процессами, в которых исполняются команды. Есть режим foreground, в котором шелл ждёт завершения процесса с командой, есть background, где шелл запускает команду и тут же читает следующую. При этом Bash умеет переводить команду из одного режима в другой, для чего там есть понятие «Job», как группа процессов, исполняющая команду. Например, пайпы собирают несколько процессов в один Job, который может быть отправлен в фон или в foreground целиком.

## 3.5. Lessons learned

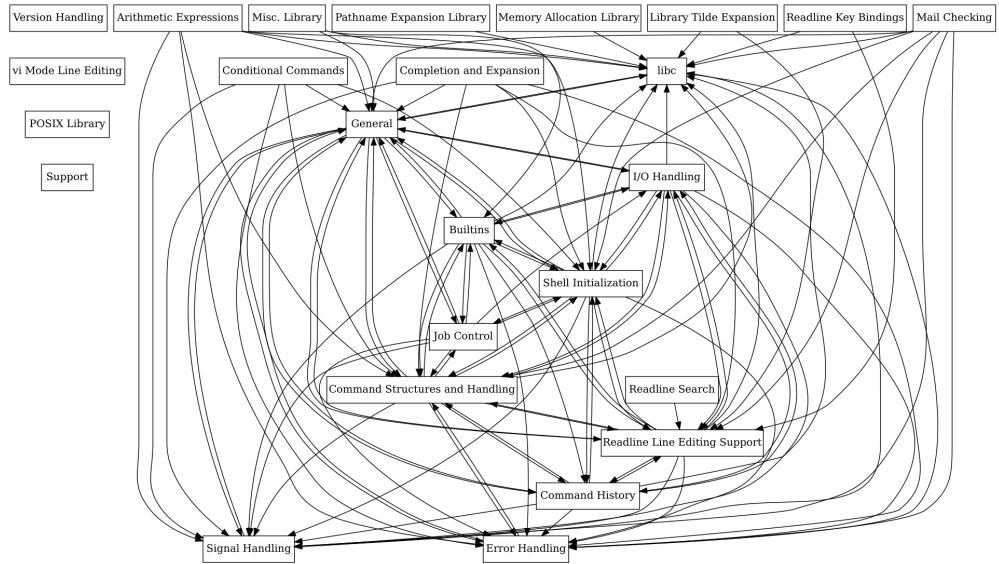
Вот кратко вещи, на которые разработчик Bash Chet Ramey обратил внимание в ходе разработки.

- Хорошие комментарии к коммитам со ссылками на багрепорты с шагами воспроизведения сильно облегчают жизнь.
- Хороший набор тестов — тоже, Bash имеет тысячи тестов, покрывающие практически всю неинтерактивную функциональность.
- Сильно помогли стандарты, как внешние на функциональность шелла, так и внутренние на код.
- Хорошая пользовательская документация важна.
- Переиспользование сильно экономит время.

## 3.6. Как обстоят дела на самом деле

Товарищи из университета Южной Калифорнии исследовали «настоящую» архитектуру Bash с целью получить «базовую» архитектуру, по которой можно было бы оценивать эффективность различных инструментов автоматического восстановления архитектуры.

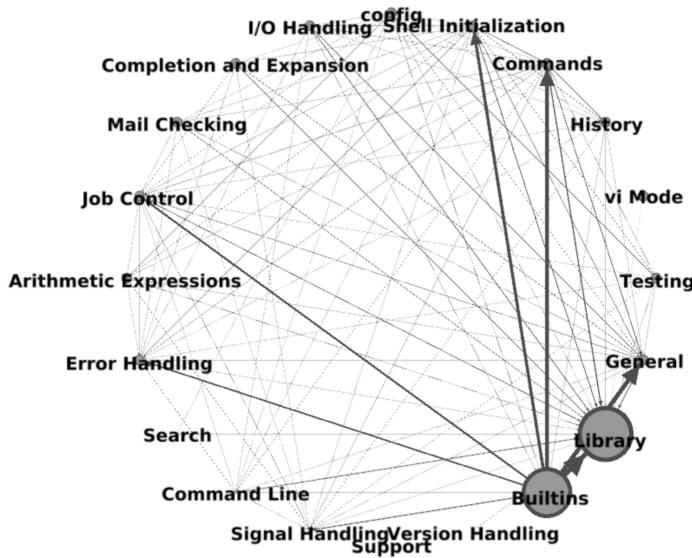
Один аспирант 80 часов копался в исходниках, после чего отправил результаты автору и тот ещё высказал свои соображения. В итоге получилась вот такая структура системы:



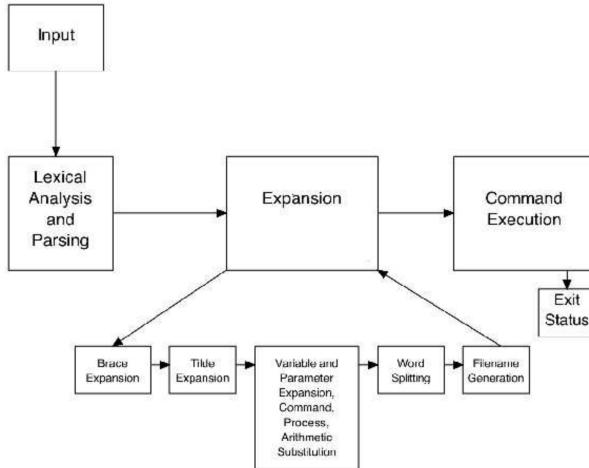
Видно, что зависимостей в коде больше, чем на картинке с концептуальной архитектурой и поток данных на структурной диаграмме совершенно неочевиден. Но некоторые схожести всё-таки есть, например, ввод-вывод реально выделяются в отдельный кластер компонентов.

Bash имеет размер порядка 70К строк кода, около 200 отдельных файлов. В ходе восстановления архитектуры было выделено 25 компонентов, из которых 16 относятся к ядру функциональности системы, 9 — вспомогательные. Выяснилось, что структура папок практически не соответствует компонентам, только два компонента имеют свои отдельные папки в исходниках.

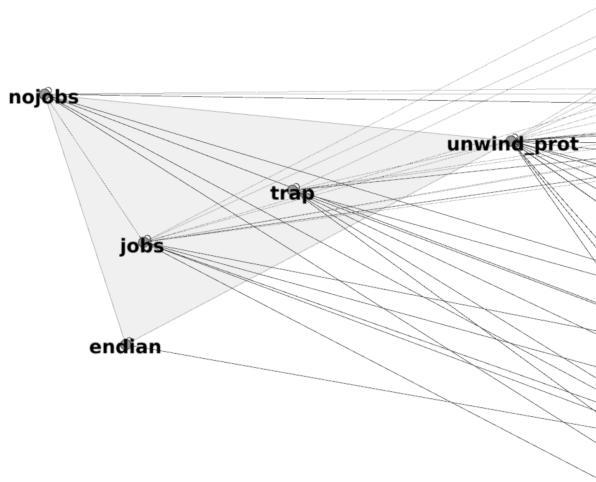
Вот автоматически восстановленная по исходникам архитектура системы, с компонентами и зависимостями:



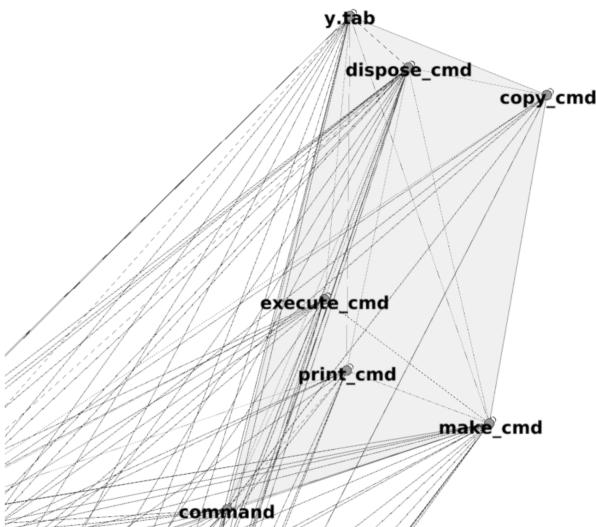
Сравните с исходной концептуальной архитектурой:



Это должно ещё раз проиллюстрировать мысль касательно архитектур реальных приложений — есть архитектура как она проектировалась (prescriptive architecture), есть архитектура как она реализована в коде приложения (descriptive architecture), и в ходе эволюции приложения эти архитектуры становятся всё больше и больше непохожими друг на друга. Эти расхождения связаны с понятиями «architectural drift» (привнесение в архитектуру вещей, которых в исходной архитектуре не было, без нарушения принципов исходной архитектуры) и «architectural erosion» (привнесение в реализацию нарушений принципов исходной архитектуры). Для долгоживущих систем архитектурная эрозия становится довольно критичным фактором, приводящим к тому, что исходное разбиение на компоненты перестаёт быть валидным вообще. Bash в этом плане довольно показателен, поскольку ему много лет. Вот увеличенный вид компоненты управления Job-ами:



Как видим, каждая сущность с этой компоненте больше связана с внешними сущностями, чем с сущностями внутри компоненты, то есть coupling очень высок, а cohesion, судя по всему, очень низок. С командами дела обстоят ещё хуже:



Собственно, поэтому важна архитектурная документация и нелишне наличие архитектора, который следил бы за тем, чтобы код и документация не очень расходились. Иначе приложение быстро превратится в гигантский ком кода, где всё зависит от всего и ломается при любом изменении.

## 4. Git

### 4.1. Ключевые требования

Git, как известно, распределённая система контроля версий. Появился он в результате грустной истории с ядром Linux — оно некоторое время хранилось в проприетарной распределённой системе BitKeeper (2000 год, кстати, одна из первых), но потом разработчики ядра что-то не поделили с компанией-производителем (BitMover), и им пригрозили отзывом лицензии. Линусу Торвальдсу пришлось срочно разрабатывать новую систему контроля версий (потому что ему не нравилась система CVS, в которой так же хранились исходники Linux в то время). Поначалу это был просто набор скриптов на bash для управления патчами, приходившими по почте. С самого начала основными требованиями к разработке были следующие.

- Распределённая разработка с тысячей коммитеров — так же, как в BitKeeper и не так, как в CVS, должны были быть поддержаны процессы, при которых каждый участник может работать у себя локально, не мешая ничьей работе и сам определяя, когда его часть готова к публикации. Для большого проекта с открытым исходным кодом, надо которым работают тысячи никому ничего не должных энтузиастов, это необходимо.
- Защита от порчи исходников — возможность отменить мердж, смердиться вручную. Опять-таки, тысячи энтузиастов, которые никому ничего не должны, и половина из которых вообще толком программировать не умеет.
- Высокая скорость работы — речь идёт о сотнях тысяч коммитов всё-таки.

Несколько иронично то, что BitKeeper сам в 2016 году стал опенсорсным, и при этом не то чтобы очень популярен. Никогда не ссорьтесь с opensource-сообществом по поводу лицензий на ПО.

### 4.2. Представление репозитория

Когда мы набираем `git init`, создаётся папка `.git`, где лежит вся информация гитового репозитория. Она имеет следующую структуру:

- **HEAD** — ссылка на текущую ветку, которую зачекаутили в рабочей папке;
- **index** — staging area, то место, где формируется информация о текущем коммите;
- **config** — конфигурационные опции гита для этого репозитория;
- **description** — «is only used by the GitWeb program, so don't worry about it» (c) Git Book;
- **hooks/** — хук-скрипты (возможность выполнить произвольный код при каком-то действии типа коммита);
- **info/** — тоже локальные настройки репозитория, сюда можно вписать игнорируемые файлы, которые вы не хотите писать в `.gitignore`, чтобы их не коммитить;

- **objects/** — самое интересное, тут лежит собственно то, что хранится в репозитории;
- **refs/** — тут лежат указатели на объекты из objects (ветки, как мы увидим в дальнейшем);
- ... — прочие штуки, которые появляются в процессе жизни репозитория и нам пока не интересны.

Гит вообще появился как набор утилит, которые позволяют быстро сделать систему контроля версий, а не как полноценная система контроля версий, так что у гита, помимо общезвестных команд, есть и команды, позволяющие напрямую работать с репозиторием и делать с ним вручную ужасные вещи. Сам по себе репозиторий в гите — это просто хештаблица, которая отображает SHA-1-хеш файла в содержимое файла, ничего более. Можно класть в неё объекты (даже не обязательно файлы), можно получать. Например, вот так:

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack

$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4

$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Создали пустой репозиторий, гит нам создал структуру папок `.git/objects`, пока пустую. Командой `git hash-object` мы положили в репозиторий новый объект — строчку `'test content'`. Ключ `-w` означает, что надо не просто посчитать хеш объекта, но и реально записать его на диск, ключ `--stdin` означает, что содержимое объекта надо получить из входного потока, а не из файла. Вызов этой команды вернул нам SHA-1-хеш того, что получилось, и заодно создал файл на диске с содержимым, положив его в `.git/objects`, в подпапку, называющуюся как первые два символа хеша, и в файл, называющийся как остальные 38 символов хеша.

Как достать то, что мы сохранили, обратно:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Команда `git cat-file` показывает содержимое файла, ключ `-p` говорит определить тип объекта и красиво показать его содержимое.

Уже можно сделать версионный контроль вручную с использованием рассмотренных команд (правда, для этого нам потребуется настоящий файл, версионировать строку, как в предыдущем примере, не интересно):

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30

$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1

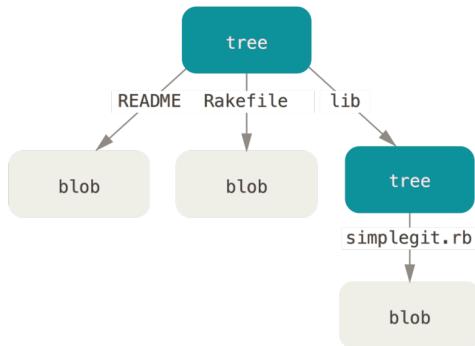
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Каждая новая версия в данном случае хранится как отдельный объект, но всему своё время.

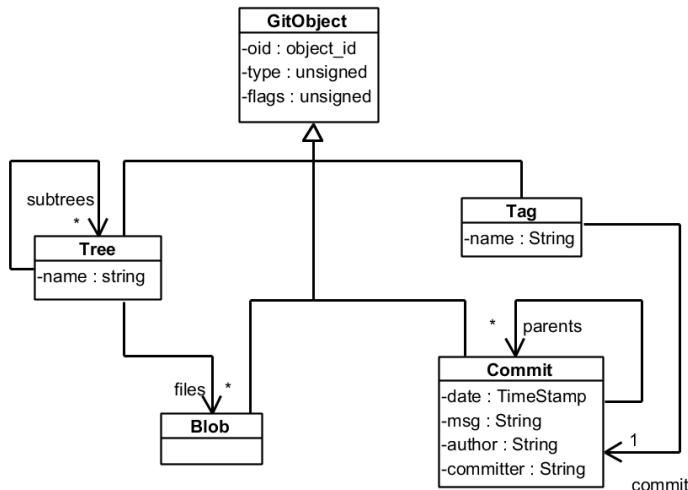
Объект, кстати, называется «blob» (Binary Large OBject), и он хранит только данные, так что даже имя файла в нём не хранится, а, наверное, хотелось бы. За хранение имени файла, а также за хранение папок и вообще иерархии объектов отвечает объект «tree». Например, вот так могло бы выглядеть дерево, на которое указывает коммит master в некотором репозитории (два файла и одно поддерево):

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

Синтаксис `master^{tree}` говорит, что надо отобразить master как tree-объект, а не как commit-объект. Вот так можно себе представить дерево, приведённое в примере:



Вот примерная UML-диаграмма классов всех объектов, которые могут находиться в готовом репозитории:



Все они являются объектами, поэтому имеют свой SHA-1-хеш, тип, который позволяет их отличить друг от друга, размер и данные. Blob и Tree мы уже видели, Tree содержит в себе поддеревья и Blob-ы. Осталось разобраться с коммитами и тэгами.

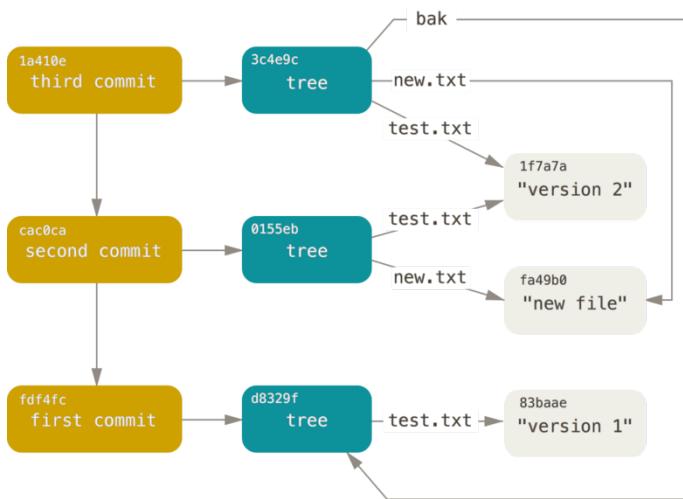
Коммиты нужны для хранения метаинформации — кто сделал изменение, когда и почему. Дерево ничего такого не хранит, в этом смысле оно напоминает узел файловой системы (в UNIX-подобных системах распространён термин inode), так что на объекты из дерева ссылаются коммит-объекты. Вот так это выглядит:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700
```

## first commit

Ещё, что не показано на картинке, но тоже есть — коммит хранит список коммитов-родителей, но вообще понятие «родитель» для коммита связано с ветками, поэтому про них чуть попозже. Вот, наверное, знакомая картинка про то, как коммиты можно представлять себе в виде указателей на узлы дерева в базе:



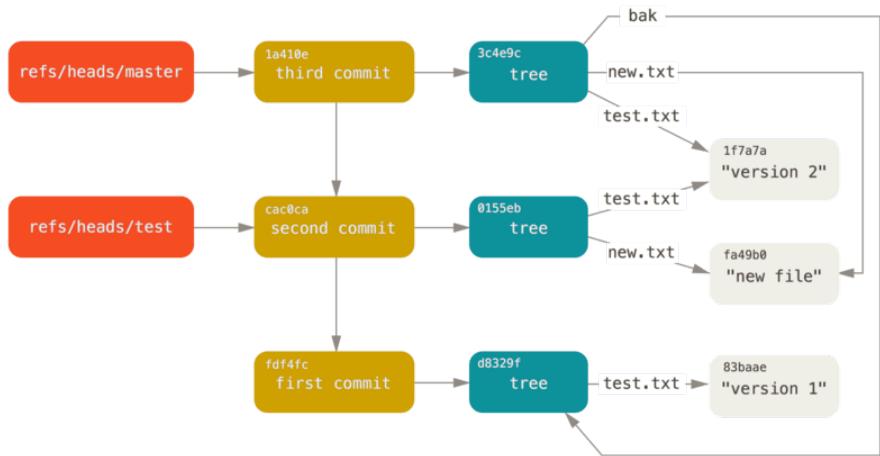
Теперь у нас есть объекты, хранящие в себе содержимое файлов (blob-ы), объекты, хранящие в себе структуру файлов и их имена (tree-объекты), объекты, хранящие в себе информацию об истории модификаций первых двух видов объектов, и уже, в принципе, система контроля версий могла бы получиться. Но пользоваться ей было бы очень неудобно, потому что каждый объект идентифицируется только своим SHA-1-хешем, и чтобы делать что-нибудь содержательное, надо было бы эти хеши помнить. Чтобы с этим помочь, придуманы references. Reference — это просто ссылка на коммит. Reference даже не объект, это просто файл, внутри которого лежит SHA-1-хеш объекта из базы. При этом reference-ы бывают двух типов — head-ы и tag-и. Они хранятся в папке .git/refs, .git/refs/heads и .git/refs/tags соответственно. Мы можем сделать свою собственную ветку, создав сами такой файл:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Совсем вручную это делать можно, но не принято, есть команда `git update-ref`, которая, во-первых, проверяет, что ref создаётся в правильной папке, во-вторых, заносит действие с reference в так называемый reflog, про который тоже чуть попозже, но вообще

— это штука, которая помнит, что происходило со ссылками и может помочь востановить случайно удалённую ветку. Традиционная картинка, поясняющая суть ссылок:



Среди всех ссылок выделяется самая главная, та, которая соответствует ветке, лежащей сейчас в рабочей копии. Она внезапно хранится не в `.git/refs`, а прямо в корне папки `.git`, в файле, который называется `HEAD`. Причём это даже не ссылка, а символическая ссылка, то есть ссылка на ссылку:

```
$ cat .git/HEAD
ref: refs/heads/master

$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

Команда `git symbolic-ref` нужна для «вежливого» обновления символьской ссылки, которая проверяет корректность того, что происходит. Таким нехитрым образом можно переключаться между ветками, но обратите внимание, что `index` ничего про это не знает, так что файлы из старой ветки будут считаться добавленными к коммиту, потому что они были в её индексе и никто их оттуда не убрал. Так что `git checkout` всё-таки не только обновляет `HEAD`.

Последний из объектов, который надо рассмотреть — это тэги. Тэг — это просто указатель на коммит. Ну, на самом деле, не всё так просто, потому что мы видели его на диаграмме с объектами в базе, а `reference` — не объект. Дело в том, что тэги бывают двух типов — легковесные и аннотированные. Легковесный тэг — это просто ссылка на коммит, которая никогда никем не двигается (её можно продвинуть вручную, но это плохо, поскольку тогда у людей, имеющих копии вашего репозитория, тэги могут начать не совпадать). Аннотированный тэг — это уже полноценный объект, который указывает на коммит, и нужен он для того, чтобы иметь возможность добавить к тэгу разную метаинформацию типа автора, сообщения и даты.

Пример, как сделать вручную легковесный тэг:

```
git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

А вот аннотированный тэг и как он хранится:

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700
```

```
test tag
```

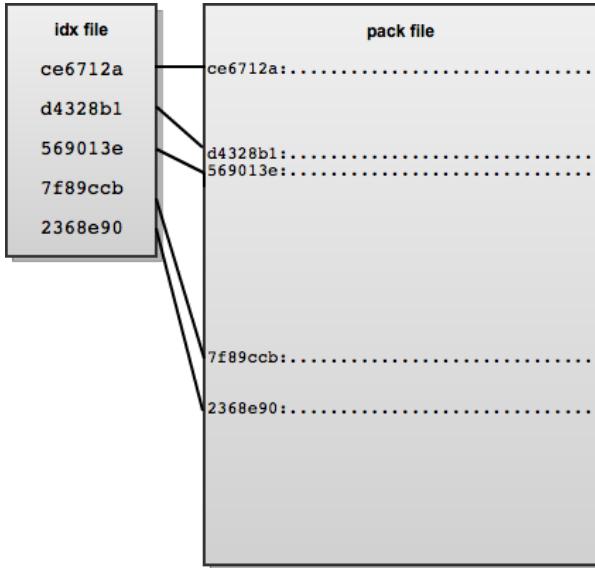
## 4.3. Pack-файлы

Казалось бы, теперь всё, но тут мы вспоминаем, что все объекты в репозитории всё ещё хранятся целиком, так что если у нас есть длиннющий исходник и мы в нём поменяли одну строчку, у нас получится два длиннющих исходника. Самое удивительное, что, в общем-то, в гите поначалу так и есть, репозиторий некоторое время просто раскопирует изменённые файлы. Естественно, файлы сжимаются zlib-ом, так что занимают чуть меньше места, чем могли бы, но всё равно, для системы контроля версий такая ситуация довольно странна. На помощь приходят pack-файлы:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/978e03944f5c581011e6998cd0e9e30000905586.pack
```

Тут мы выполнили команду `git gc` (Garbage Collect), в результате которой некоторые «нормальные» объекты удалились (на самом деле, все кроме «висячих», то есть недостижимых по ссылкам) и появилось два файла: `.idx` и `.pack`. Второй файл содержит упакованными все наши объекты, и тут уже применяется дельта-компрессия, причём, что интересно, последняя версия файла хранится целиком, а предыдущие версии — как дельты относительно более свежей версии, то есть как бы «назад» (что логично, скорее всего, последняя версия нужна чаще). Первый файл — это оглавление для второго файла, именно его передают по сети, когда делается `git push/git pull` и локальный или удалённый гит пытаются понять, какой информации у него нету. Вот так примерно это выглядит:



Упаковка объектов в .pack-файлы происходит, когда:

- выполняется git push;
- слишком много «свободных» объектов (порядка 7000);
- вручную вызвана git gc.

Если pack-файл уже есть, то новые объекты могут упаковаться в новый файл, оставив старый неизменённым, а может произойти перепаковка и несколько .pack-файлов будут слиты в один (важно понимать, что .pack-файлов может быть несколько и вся работа с ними скрыта от пользователя системы контроля версий). Почему всё так хитро — упаковка в .pack-файл требует пересчёта дельт и вообще очень трудоёмкая операция, так что делать её каждый коммит было бы очень раздражающе для пользователя. Есть ещё команда git gc --auto, которая проверяет, не надо ли запаковать объекты, она вызывается при каждом коммите и, как правило, ничего не делает, иногда всё-таки вызывая git gc. Внутрь pack-файла можно посмотреть командой git verify-pack, но это не то чтобы сильно полезно на практике, так что подробности в Git Book.

## 4.4. Reflog

Все нормальные команды гита записывают всё, что они делали с reference-ами, в файлы в папке logs, где, в частности, лежит лог того, что происходило со ссылкой HEAD, и его можно просмотреть командой git reflog:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

Или получить более подробную информацию командой `git log -g`:

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

    third commit
$ git branch recover-branch ab1afef
```

Если мы случайно откатили ветку и потеряли тем самым какой-то нужный коммит, можно найти его в `reflog`-е, взять его хеш и сделать на него `checkout`.

А теперь как более капитально прострелить себе ногу. Шаг 1, удаляем ветку:

```
$ git branch -D master
```

Шаг второй, сносим все логи, чтобы нельзя было восстановить ветку по SHA-1-хешу последнего коммита, на который она указывала:

```
$ rm -Rf .git/logs/
```

Казалось бы, всё, репозиторий запорот и надо делать домашку заново? Нет, если база объектов на месте, можно воспользоваться командой `git fsck --full`, которая распечатает нам все висячие объекты вместе с их хешами:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

Теперь мы можем посмотреть на них командой `git cat-file -p`, выбрать тот, который больше всего похож на последний коммит той ветки, которую мы удалили, и восстановить ветку по его хешу: `git branch recover-branch ab1afef`. Ещё позитивно то, что Git не удалит даже «висячие» объекты несколько месяцев, если его явно не попросить, несмотря на то, как расшифровывается имя команды `git gc`, так что если вы потеряли ветку, то с большой вероятностью она всё ещё где-то есть и её можно восстановить.

## 4.5. Lessons learned

Поначалу некоторые команды были реализованы как shell-скрипты, потому что так было быстрее. Однако это помешало, во-первых, интеграцию git со средствами разработки, во-вторых, существенно усложнило портирование git на Windows. Что, в общем-то, не сильно расстроило Линуса Торвальдса и окологитовое сообщество середины 2000-х, но

на самом деле негативно сказалось на внедрении git, поскольку крупные компании избегали пользоваться им из-за проблем с переносимостью. Впоследствии был реализован проект Git for Windows, где уже вся функциональность была реализована в виде разделяемой библиотеки, и дело пошло.

Ещё важный момент, происходящий из изначального дизайна git как набора инструментов для управления патчами — у него весьма хитрая система команд («plumbing», которым, в общем-то, никто не пользуется, и «porcelain»). Даже если не задумываться о том, что новички в git могут случайно нагуглить git cat-file и нескоро понять, что эти команды им знать не надо, набор команд сложноват и не то чтобы дружественен к начинающим пользователям. На первом курсе, чтобы научить студентов пользоваться git, требуется обычно две пары, плюс у них всё равно весь первый курс время от времени возникают проблемы, требующие некоторого вмешательства.

## 5. Battle for Wesnoth

Следующий пример архитектуры диаметрально противоположен консольным утилитам. Речь пойдёт про игру Battle for Wesnoth, пошаговую стратегию с открытым исходным кодом, один из относительно немногих подобных проектов, который реально игрален, развивается более 15 лет и до сих пор жив, имеет 93% позитивных отзывов (из более 2700) на Steam. При этом игра распространяется бесплатно и имеет порядка 4 миллионов скачиваний, включая скачивания со Steam и официального сайта. Разработка началась в 2003 году, игра написана на C++ и на данный момент имеет кодовую базу порядка 200000 строк кода, плюс порядка 250000 строк декларативного описания контента на их собственном предметно-ориентированном языке.

### 5.1. Architectural Drivers

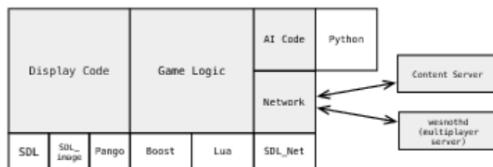
Вся архитектура игры строится вокруг одного принципа — доступности для новых разработчиков и авторов контента. Для некоммерческого open-source-проекта возможность привлекать сторонних авторов, быстро вводить их в курс дела и давать возможность создавать контент или править движок игры — критична для выживания. Как обычно, речь идёт о проекте, где никто никому ничего не обязан, который живёт достаточно долго, чтобы несколько поколений maintainer-ов успели потерять интерес, где, в отличие от систем, рассмотренных ранее, большая часть интересующихся проектом людей — скажем так, не очень зрелого возраста, и в подавляющем большинстве без технического образования.

Поэтому основными требованиями, легшими в основу архитектуры, стали:

- возможность декларативно создавать контент (кампании, юниты, карты, сюжет и т.д.) на предельно простом предметно-ориентированном языке;
- использование широко распространённых библиотек, минимизация внешних зависимостей;
- простота кода, возможно, в ущерб технической красоте;
- кроссплатформенность.

## 5.2. Высокоуровневая архитектура

Диаграмма «компонентов» проекта представлена на рисунке:



Опять-таки, диаграмма из квадратиков и стрелочек вместо формальных нотаций UML (чтобы не смущать юных кодеров). И, кстати, эта мутная картинка — лучшее, что удалось найти про высокоуровневую архитектуру системы (и то в AOSA, не в каком-нибудь диздоке на сайте проекта, а вики на гитхабе у них вообще отсутствует). Это, на мой взгляд, несколько противоречит идею о быстром и простом введении новых разработчиков в проект, но увы.

Итак, на диаграмме белым выделены внешние библиотеки, серым — то, что непосредственно относится к проекту. Стрелочками с основной игрой соединены отдельные программы:

- wesnothd — сервер для многопользовательской игры;
- content server — репозиторий с кампаниями, картами и другим дополнительным контентом, с помощью которого прямо из игры можно качать расширения.

Ввод-вывод и отрисовка графики выполняются библиотекой SDL (Simple Directmedia Layer), очень известной в игровой индустрии. Она умеет работу с клавиатурой/мышью, отображение двумерной графики, звук, портирована на все нормальные и не вполне нормальные платформы (например, Android, так что SDL-приложения, скорее всего, даром запустятся там, но без тюнинга пользовательского интерфейса будут неиграбельны).

Boost используется как библиотека общего назначения, стандартная для практически любого проекта на C++, она обеспечивает кучу полезных вещей, которые постепенно перекочёвывают в стандартную библиотеку, но до C++11 программировать без Boost на плюсах было упражнением в мазохизме (если вы не использовали какие-то другие подобные библиотеки, типа Qt).

Pango + Cairo — для отображения текста с учётом локализации, zlib — стандарт де-факто, если надо что-нибудь архивировать, Lua и Python для скриптования, GNU gettext для интернационализации.

Сама система состоит из следующих компонентов.

- Парсер и препроцессор WML (Wesnoth Markup Language) — тот самый DSL, на котором описывается весь контент игры.
- Базовый ввод-вывод — видео, звук, сеть поверх SDL.
- GUI — виджеты пользовательского интерфейса (всякие кнопки, текстовые поля, полосы прокрутки и т.д.).

- Display module — игровая сцена, юниты, анимация и т.д. Это не GUI, потому что отображение сцены всё-таки отдельная большая задача, большая часть специфичного для системы кода относится именно к игровой сцене.
- Модуль ИИ — думаю, понятно (хотя, наверное в наше время стоит особо отметить, что в играх ИИ — это не про нейросети).
- Поиск пути — модуль для поддержки ИИ, содержит, помимо собственно, алгоритмов поиска пути, утилиты для работы с гексагональной доской, поскольку это тоже не очень тривиальная задача.
- Генератор карт — не все карты рукодельные, есть и развитый механизм случайной генерации.
- Специализированные модули — модуль титульного экрана, Storyline module (для проигрывания катсцен по ходу кампании), «Play game» module (управление основным игровым процессом).

### 5.3. Wesnoth Markup Language

Wesnoth Markup Language — язык для описания контента. Изначально планировалось использовать XML-описания, но авторы решили ориентироваться на контентоделов, для которых даже HTML или Python кажутся сложными и породили следующее:

```
[unit_type]
id=Elvish Fighter
name= _ "Elvish Fighter"
image="units/elves-wood/fighter.png"
hitpoints=33
advances_to=Elvish Captain,Elvish Hero
{LESS_NIMBLE_ELF}
[attack]
name=sword
icon=attacks/sword-elven.png
range=melee
damage=5
[/attack]
[/unit_type]
```

В принципе, очень похоже на XML, но, как видно, атрибуты и дочерние тэги синтаксически не очень различаются, и атрибуты выглядят несколько более симпатично, чем в XML (хотя бы без лишних кавычек). К тому же, нет всяких неймспейсов и тэги выглядят менее «агрессивно». Кроме того, есть препроцессор — например, директива `{LESS_NIMBLE_ELF}` тут используется для подстановки стандартных для эльфов параметров.

Вообще, препроцессор довольно развитый, умеет в параметры макросов, условные операторы и т.п. Например,

```

#define GOLD EASY_AMOUNT NORMAL_AMOUNT HARD_AMOUNT
    #ifdef EASY
    gold={EASY_AMOUNT}
    #endif
    #ifdef NORMAL
    gold={NORMAL_AMOUNT}
    #endif
    #ifdef HARD
    gold={HARD_AMOUNT}
    #endif
#endif
...
{GOLD 50 100 200}

```

Тут описывается макрос с тремя параметрами, который в зависимости от настроек сложности кампании выбирает одно из трёх переданных ему значений. {GOLD 50 100 200} — вызов макроса где-то в конфиге, который, видимо, даёт ИИ меньше денег, если игрок выбрал более лёгкий уровень сложности.

WML-описания контента разбиты по разным файлам, при загрузке уровня они все сливаются в один гигантский WML-документ, препроцессируются и загружаются в модель данных в программе. При этом при смене опций модель данных перепропроцессируется и перезагружается заново (как раз из-за условных операторов в препроцессоре). Чтобы это не было убийственно для производительности (на некоторых машинах загрузка документа занимала до минуты), применяются всякие хаки на уровне препроцессора, например, определение своей переменной препроцессора для каждой кампании и `#ifdef`-ы, окружающие специфичный для кампании контент, чтобы даже не рассматривать его при загрузке других кампаний. Кроме того, WML-документы кешируются.

В качестве хранилища данных о юнитах используется класс `unit_type`. Для представления конкретного юнита на поле боя используется класс `unit`, имеющий ссылку на соответствующий `unit_type`. `unit_type`-ы грузятся при загрузки WML в глобальную таблицу типов юнитов (это что-то вроде стиля «Knowledge Layer», `unit_type` декларативно описывает, что можно делать с юнитами, `unit` представляет сущность «операционного» уровня).

ИИ смотрит на `unit` и `unit_type` и выбирает поведение юнита исходя из флагов, прописанных в WML, и его текущего состояния. В WML нет никакой возможности описать логику действий юнита (даже декларативно), всё, что можно сделать, это ставить флаги. Например, «skirmisher» скажет ИИ и логике игры, что юнит может свободно перемещаться вблизи от юнитов врага, а не заканчивать ход, как обычные юниты. Это осознанное решение, поскольку задание поведения юнита в императивном стиле, безусловно, серьёзно расширило бы возможности контентоделов, но существенно усложнило бы описание контента. Поэтому, исходя из architectural drivers, от этой идеи отказались. Всё поведение ИИ и все возможности юнитов должны быть поддержаны в коде.

Такая же идея стоит за классом `attack_type`, описывающим возможные типы атак. Есть фиксированный набор видов атаки и эффектов, поддерживаемых движком (например, дальняя атака, атака ближнего боя, магическая атака), можно из базовых атрибутов собирать атаку конкретного вида оружия (например, атака мечом — 4 попытки нанести урон от 1 до 8 холодным оружием) и давать атаки юнитам (может, несколько), при этом игрок

может выбрать, какую атаку использовать в конкретной ситуации (например, лучники эффективны против юнитов чисто ближнего боя, потому что последние не могут ответить на дальнюю атаку).

Кроме того, для большей «ролеплейности» игрового процесса, каждому юниту при создании назначаются случайные особенности. Например, сильный юнит сильнее атакует в ближнем бою, умный юнит требует меньше опыта для прокачки. Да, каждый юнит имеет дерево прокачки с преобразованием в другой тип юнита (то самое `advances_to=Elvish Captain,Elvish Hero` в WML). При этом у каждого юнита есть инвентарь, куда можно добавить вещи, которые ещё как-то модифицируют параметры юнита. Если вспомнить, что речь идёт о стратегии, а не о RPG, звучит неплохо.

## 5.4. Мультиплеер

Архитектура многопользовательской игры заслуживает отдельного упоминания. Сервер работает по протоколу TCP/IP и реализован максимально просто: есть начальное состояние игры, которое при создании игры рассыпается всем подключённым клиентам, и пользовательские команды. При выполнении хода игрока команды сериализуются и рассыпаются всем подключённым клиентам, и при этом сохраняются в логе игры на сервере. Клиенты проигрывают команды, тем поддерживая синхронность изменений игрового мира. При этом новый клиент может подключиться прямо в процессе игры, ему пошлют начальное состояние и все команды из лога, так что он сможет восстановить текущее состояние игры. Это же даёт бесплатно возможность просматривать повтор игры.

Сервер просто пересыпает команды между клиентами, никакой игровой логики на сервере нет. Нет и никакой защиты от читов, и это тоже осознанное архитектурное решение — авторы решили, что в Wesnoth не должны устраивать соревновательные матчи, все игры должны быть дружескими и для удовольствия, а не для победы. Потому посыпать некорректные с точки зрения игровой логики команды можно, сервер не возражает, но ожидается, что психически здоровые пользователи не будут этого делать. Единственное, что проверяется при подключении — версии клиентов, если они не сойдутся, то проигрывание команд может привести к разным результатам, так что клиенту с неправильной версией отказывают в подключении.

## 5.5. Lessons Learned

Итого, результатами разработки WML и выбранной стратегии максимального упрощения стало то, что сейчас кода на WML больше, чем кода на C++ (порядка 250К строк кода WML против 200К кода на C++). К игре сделали сотни пользовательских кампаний, и бесчисленное количество других ресурсов. Репозиторий имеет более 77 тысяч коммитов, более 200 контрибьюторов, более 600 форков. Так что можно считать, что цели проекта (выжить и развиваться в условиях, когда типичному пользователю 12 лет) полностью достигнуты.

Сами разработчики в AOSA в секции Lessons Learned рассказывают больше про то, как крут их проект а не про то, почему они научились. Единственная самокритика заключается в том, что WML кажется самим же разработчикам весьма убогим и неэффективным. Однако же успех проекта показал, что технически неправильное решение избрести велосипед оказалось правильным в плане возможностей по привлечению авторов контента. В целом,

авторы делают вывод, что задача сделать и контент, и код доступными для модификации приходящими в проект людьми, особенно без богатого опыта разработки, очень сложна. Я бы сказал, в плане движка игры они и не пытались особо, но может, это и к лучшему, чтобы уж совсем тупые script kiddies не отвлекали своими пуллреквестами нормальных разработчиков.