Конспект по проектированию ПО

v0.2

Содержание

0 Disclaimer

Это конспект по курсу «Проектирование программного обеспечения», прочитанному year2011 на четвёртом курсе. Автор конспекта была на всех парах, на которых был и преподаватель[1], кроме одной (на которой, по счастливому стечению обстоятельств, лекции не было). В этом документе местами сказано больше, чем было на лекциях (так как он является жалкой попыткой автора подготовиться к сдаче зачёта), но местами — меньше (так как эти места были относительно лирическими отступлениями на лекциях, которые в общем и в целом вписываются в тему, но в ответе на вопрос смотрятся не к месту), так что его можно считать отрефакторенным конспектом, не покрытым тестами[2]. Дословная точность цитат не гарантируется, но никаких распятых мальчиков тут нет.
Конспект лежит по адресу `https://github.com/katyatitkova/software-design-outline`.
TODO:

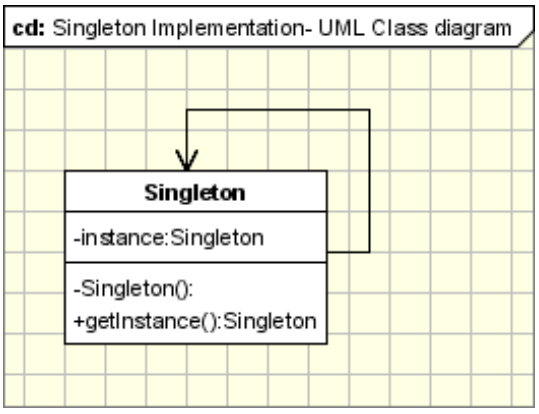
- вычитывание теми, кто был на парах;
- недостающие картинки к паттернам;
- дописать ещё про паттерны;
- убрать пробелы в многослойной архитектуре.

1 Порождающие паттерны[3]

Книжка по паттернам — конечно же, Банда Четырёх[4]. Картинки здесь — с `http://www.oodesign.com`.

1.1 Singleton

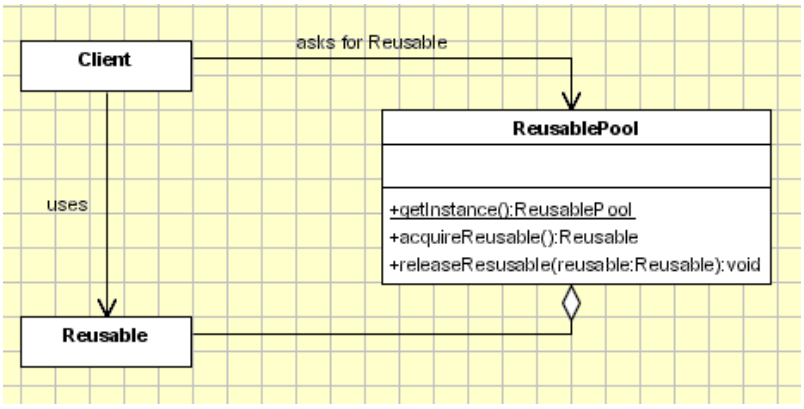
Гарантирует, что у класса есть только один экземпляр, и предоставляет ему глобальную точку доступа.



[1] Дмитрий Юрьевич Кочелаев
[2] Если вы не поняли последние пять слов, то поймёте после прочтения шестой главы.
[3] На самом деле, здесь почти всё взято из книжки.
[4] Ralph Johnson, John Vlissides, Richard Helm, and Erich Gamma “Design Patterns: Elements of Reusable Object-Oriented Software”

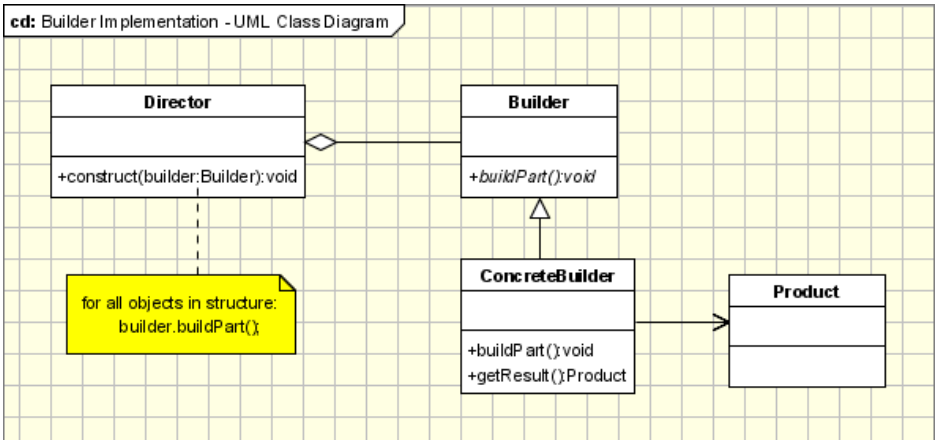
1.2 Pool[5]

Храним объекты готовыми к использованию: когда нужен, берём из пула (а не создаём), когда не нужен — возвращаем в пул (а не удаляем). Pool объектов никогда нельзя писать самому.



1.3 Builder

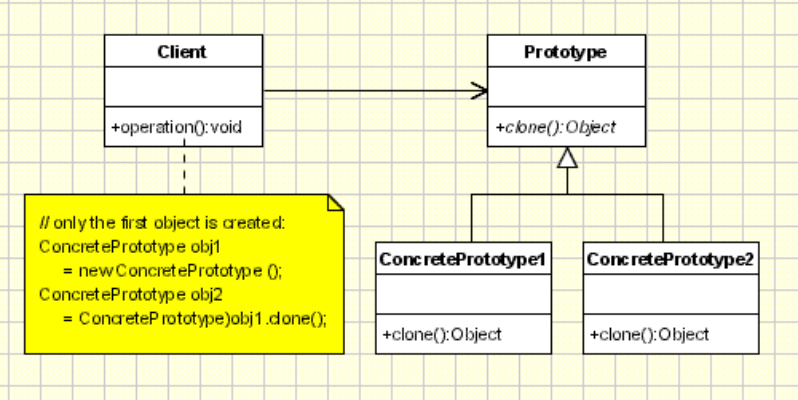
Отделяет конструирование сложного объекта от его представления: в результате одного и того же процесса конструирования могут получаться разные представления.



1.4 Prototype

Задаёт виды создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты, копируя этот прототип.
Используем, когда нужно много одинаковых объектов: например, в текстовом редакторе отрендерили один раз букву и юзаем. А ещё существует извращённый прототип с наследованием.

[5] Нет в GoF.

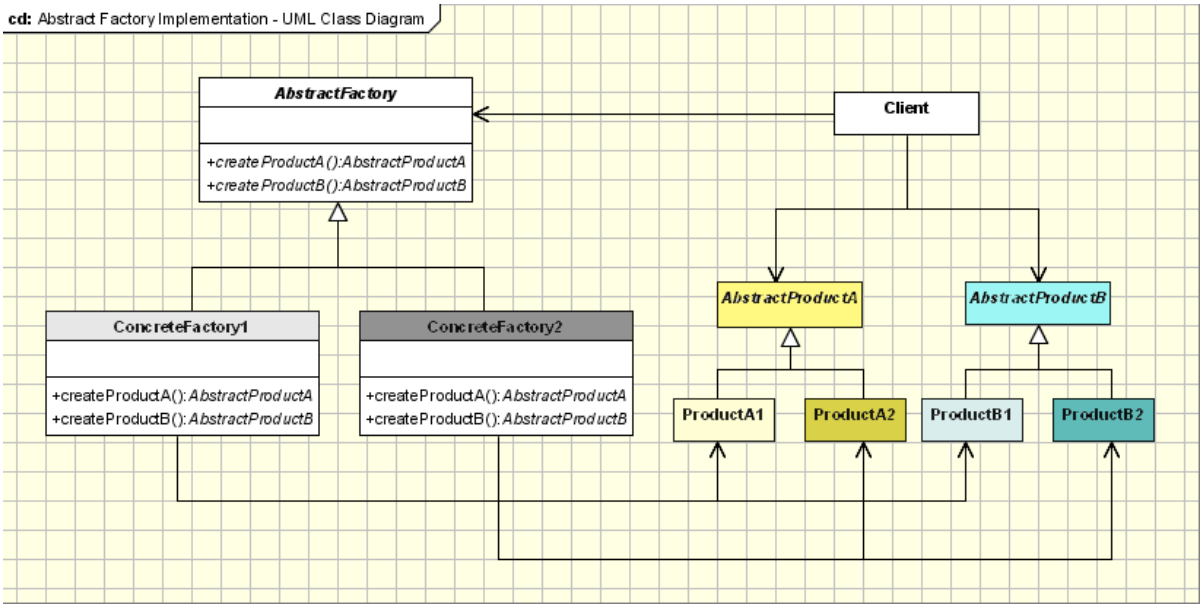


1.5 Factory Method

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Позволяет делегировать инстанцирование подклассам.

1.6 Abstract Factory

Фабрика фабрик. Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

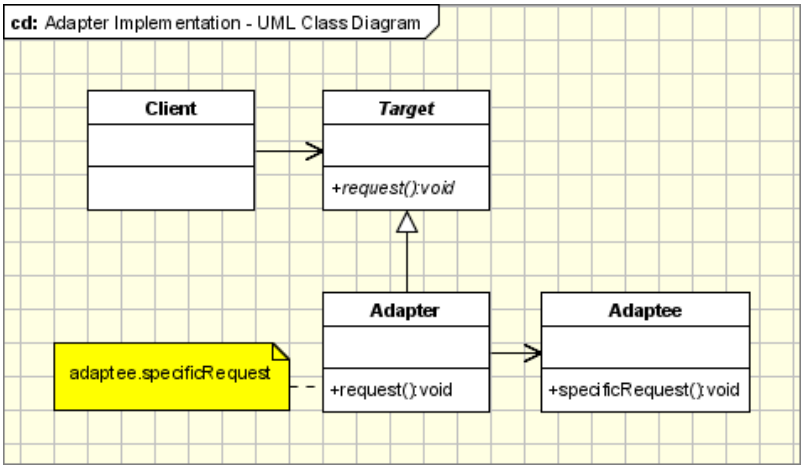


2 Структурные паттерны[6]

2.1 Adapter

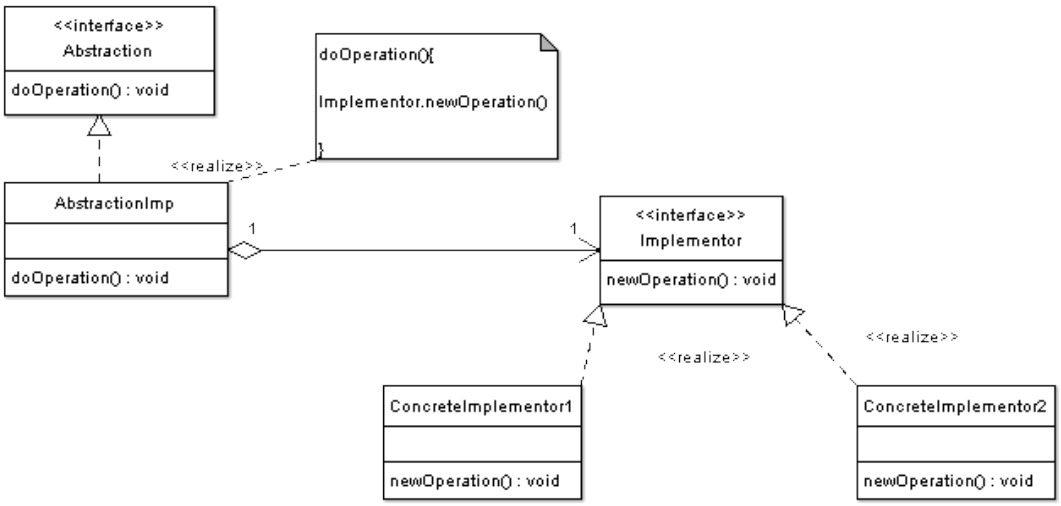
Чтоб скрестить ежа с ужом: для юзания классов, не совместимых по интерфейсу.

[6] И тут тоже.



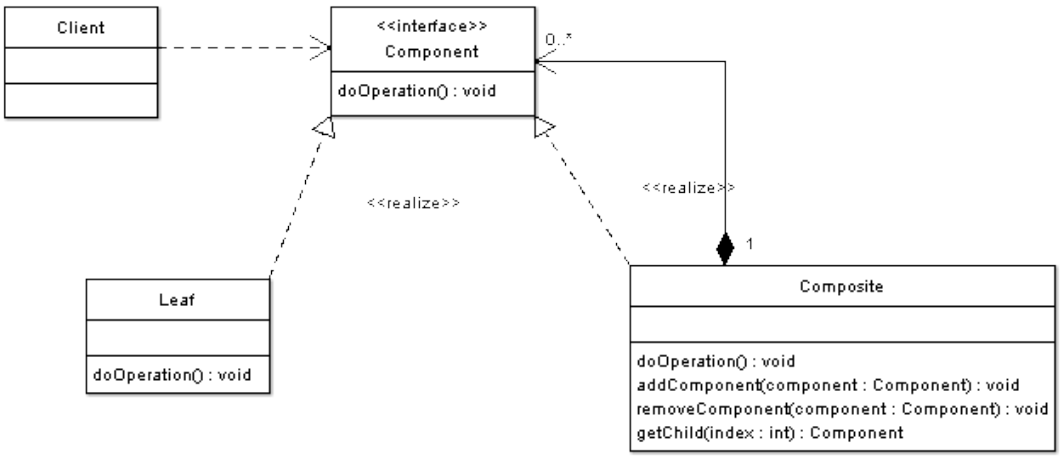
2.2 Bridge

Отделяет абстракцию от её реализации так, чтобы их можно было изменять независимо.



2.3 Composite

Компонует объекты в древовидные структуры для представления иерархий часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

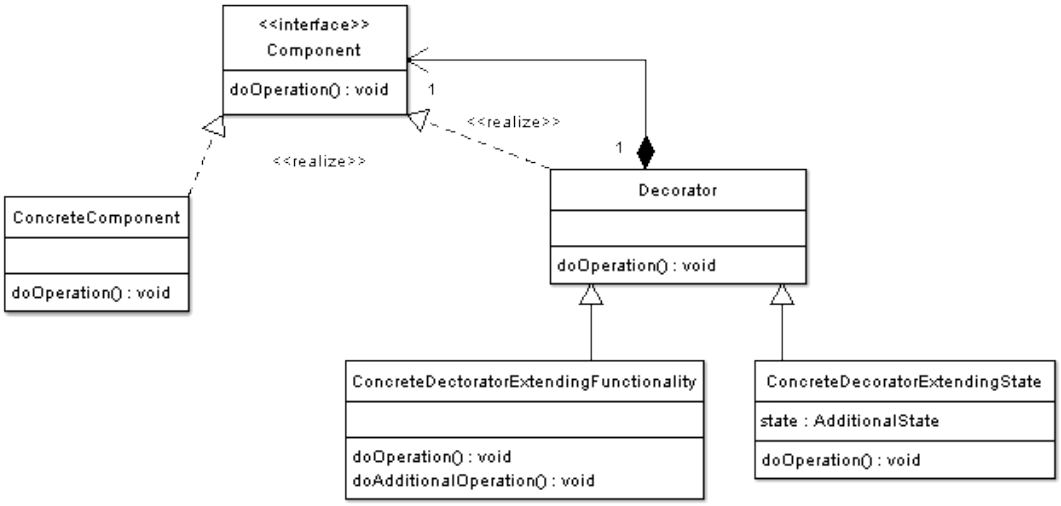


2.4 Facade

Предоставляет унифицированный интерфейс вместо набора интерфейсов подсистемы. Определяет интерфейс более высокого уровня для упрощения использования подсистемы. Если нет планов переписать, расширить, etc., то фасад не нужен.

2.5 Decorator

Динамически добавляет объекту новые обязанности. Является альтернативой порождению подклассов с целью расширения функциональности.



3 Многослойная архитектура

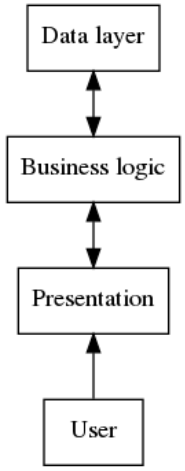


Рис. 1: Классика

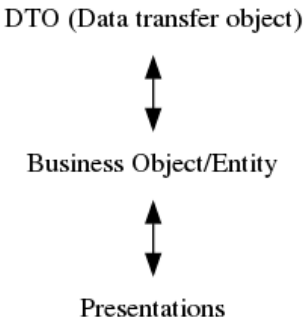


Рис. 2: 1 layer — 1 object

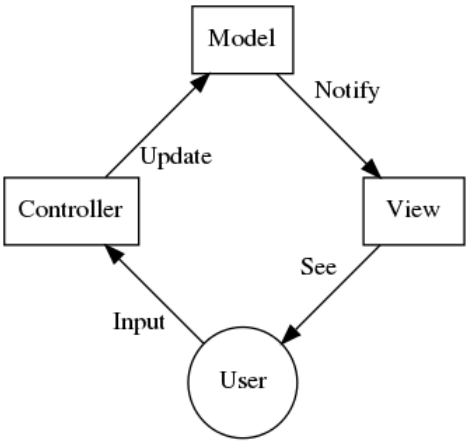


Рис. 3: MVC

Казалось бы, по классической картинке всё понятно. Business logic может разрастаться вертикально.

1. 1 layer — 1 object

   DTO — это объект, который используется для передачи данных между слоями. Сокращает количество запросов, которые обычно дороги.

2. All layers — 1 object[7]

Дальше было что-то про ORM.
MVC (Model-view-controller)

4 Inversion of Control & Dependency Injection

Реализация принципа "Don't call us, we'll call you". Например, была консольная программка, которая складывала числа, которая говорила: «введите a», «введите b» (то есть программа говорила, что делать), а теперь — программка с полями для ввода и кнопкой calculate, где можно править значения в любом порядке (то есть мы управляем).
IoC нужен, чтобы:

1. разделить функциональность между этапами запуска и реализации;
2. разграничить: каждый модуль занимается только своей задачей, физически нельзя залезть в чужое;
3. дать модулям полагаться на контракты.

Способы реализации:

[7]Кажется, никто из тех, кто ходил, не помнит, что это :(Читающий, если ты помнишь — допиши сам или расскажи Титковой!

1. Dependency injection — передаём dependency (сервис) dependent object'у (клиенту), то есть сервис становится частью клиента. Весело дебагать. Из интерфейса понятно, что и как у нас с клиентом.

2. Service locator — сами ищем сервис, который отвечает за нашу функциональность. Гораздо проще реализовать (в примитиве — HashMap), так как перекладывает часть логики на клиента, но значительно сложнее протестировать. В использовании зависим от интерфейса.

Подходы к конфигурации DI:

1. inline (прямо в коде);

2. configs (xml, json...);

3. annotation.

DI ещё даёт возможность использовать прокси и аспекты.
AOP (aspect-oriented programming)
Аспект — некий модуль, который реализует сквозную функциональность (функциональность, которую нельзя выделить в отдельные сущности; её реализация распределена по различным модулям программы).
Advice (совет) — дополнительная функциональность, которую хотим внести в код.
Injection point (join point) — то место, где следует применить advice.

5 Message Queue

Message queue — способ подружить два и более приложений. До этого были CORBA[8], OLE[9], WS[10].
Пример: рассылаем смс о транзакциях в банке.

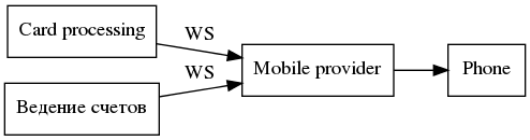


Рис. 4: Плохая схема

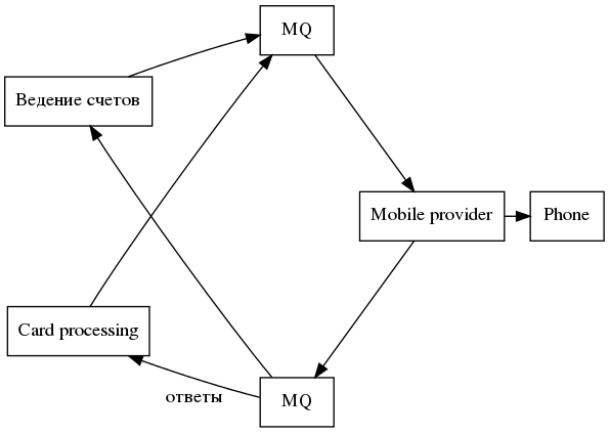


Рис. 5: Хорошая схема

У нас есть какая-то пропускная способность для отправки смсок. Первого числа всем начислили зарплату, пошёл поток смсок. При плохой схеме у нас висят транзакции, по которым ещё не отправились. При хорошей мы завершили транзакцию, отправив смску в очередь.

[8]Common Object Request Broker Architecture
[9]Object Linking and Embedding
[10]Web Service(?)

Очередь должна уметь восстанавливаться, поэтому мы делаем их три. Ответ «ок» на клиенте, когда мы положили сообщение во все очереди. Зачем три? Если одна умерла, то по тому, есть ли сосед, понимаем, мы умерли или не мы[11]
Гарантии доставки:

- at least once — доставляем минимум один раз, чтоб гарантированно получить ответ;
- at most once — доставляем максимум один раз (выполнить операцию дважды хуже, чем не выполнить вообще).

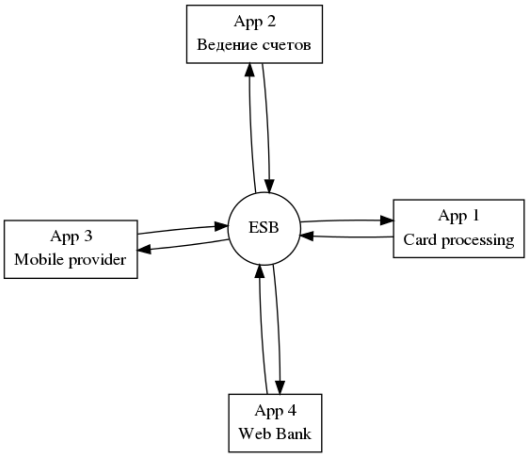


Рис. 6: (Enterprise) Service Bus. Стрелочки чаще всего — очереди

В очередях плохо хранить большие сообщения, поэтому обычно header хранят в очереди, a binary data — в file storage.

6 Рефакторинг

По рефакторингу есть классическа книжка Фаулера[12]. ДК говорит, что Фаулер считает, что «жить без рефакторинга так же сложно, как без Крыма». И советует для общего образования пролистать оглавление, а если есть куча свободного времени — почитать. Но от него хорошо засыпается[13].
Рефакторить можно код, структуру БД, интерфейсы.

6.1 Рефакторинг кода

Preconditions для рефакторинга кода с точки зрения Фаулера (не ДК):

- добавляется новая функциональность;
- исправление ошибок («как если криво висела картина, а мы заодно обои переклеили»);

[11] «Кто не смотрел “Шестое чувство”: это был спойлер»
[12] Martin Fowler “Refactoring: Improving the Design of Existing Code”
[13] . . . и поэтому автор только пролистала. Там действительно разобраны довольно очевидные проблемы с довольно очевидными путями решения.

- изучение кода («самое страшное», «как если висит полочка, вытащили её, снова переклеили обои, пролили клей на паркет, переложили его, а попутно сломали кафель в ванной. . . »)

Да, «рефакторинг — как ремонт». Вообще, «первые два пункта имеют право на жизнь», а «новая функциональность без рефакторинга — как балконы в Молдавии»[14], но «надо было переклеить одно полотнище — не надо перекладывать кафель у соседей».
Для рефакторинга нужен хороший coverage тестами, но тесты-то тоже переделаются. . .
Test-driven development, с точки зрения ДК, гиблый метод. «Есть тесты, и нет ничего. Обычно в этот момент сдавать надо». Раз сразу всё обложили тестами, то придумали и интерфейс, а он всё равно десять раз перепишется, пока будем писать код.
Кстати, выкинуть старое, написать новое — не рефакторинг.
Если у нас coverage тестами 95%, то можно не рефакторить функцию на 15 экранов. А развернуть цикл иногда ок: например, если это цикл до 4, занимающийся получением byte'ов из int'а, так понятнее будет.
Советы ДК по поводу рефакторинга кода:

- не рефакторить просто так;
- рефакторить, только если обложили всё тестами;
- сразу писать так, чтобы не рефакторить;
- полезный рефакторинг — в процессе написания.

6.2 Рефакторинг БД

Рефакторинг БД и интерфейсов — занятие мрачное и неблагодарное. Принципиальное отличие рефакторинга БД от рефакторинга кода — нет тестов для структуры БД, у всех всё может работать, но неправильно. Добавить новую колонку не страшно, обычно не ломает старое, а вот расширение поля — ой.
Советы от ДК:

- пытаться писать тесты на БД;
- спроектировать сразу хорошо.

6.3 Рефакторинг интерфейсов

После рефакторинга интерфейса все пользователи нашей библиотеки нас люто ненавидят. Правильно делать так:

1. выкатываем новый API в sandbox (в котором работают оба, можно их посравнивать между собой);
2. в продакшне работают оба API (причём хорошо сделать такой ключ, что все, кто получил его после выхода нового, не могут использовать старый);
3. можно убить старый (отвалятся примерно 15% приложений, использующих его, из них 50% ничего не заметят, 15% напишут гневное письмо, но ничего не сделают, остальные как-то приспособятся).

А за рефакторинг как самоцель надо бить.

[14] Дальше был рассказ ДК о том, что в Молдавии любят достраивать балконы вперёд, причём балконы третьего этажа опираются на достроенные балконы второго и так далее.

7 Параллельные транзакции

Перенесёмся в доисторические времена, когда VCS не было, но программисты уже существовали. Был у них шареный диск, общая папка, общие исходники. И открыл первый программист файл, и писал в него целый день. А второй открыл на два часа позже, что-то исправил, и сохранил. И первый сохранил. Но в результате у нас lost changes — никто ничего не сделал! Так вот, эти чуваки — два потока с параллельными транзакциями.
Ещё пример. Продолжаем работать, считаем замечательную метрику кода: количество классов в каждом пакете. Просто проходим по директориям и считаем. Пусть у нас есть директория `A` с классами `A1` и `A2` и директория `B` с `B1`, `B2`, `B3`, `B4`. Насчитали два класса в `A` и ушли на перекур. В это время пришёл второй чувак, создал `A3` и удалил `B3` и `B4`. Пришли, досчитали, что в `B` два класса, «сложили 2 + 2 на js, получили 22, отдаём отчёт начальству». Но проблема в том, что у нас ни в какой момент не было четырёх классов. Это — несогласованные чтения (inconsistent read).
Это лишь маленькие проблемы, которые возникают при работе без попыток что-то урегулировать. «Тут хуже, чем дикий Запад. Прям Мытищи какие-то».

7.1 Как бороться с тем, что одновременно меняем одни и те же ресурсы?

1. Isolate (изолированность транзакций): захотели поработать с файлом — делаем lock файла, «ушли, забыли разлочить, заболели на две недели, уехали в отпуск ещё на три, попутно нас прокляли»;
2. Immutability: большое количество файлов, которые неизменяемы;
3. Общаемся с данными только read-only.

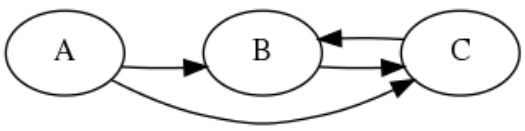
Если бы всё было read-only, проблем бы не было.

7.2 Пессимистические и оптимистические блокировки

Пессимистическая — только бы ничего не случилось. Лочим всегда, даже если не факт, что случится конфликт: залочили ресурс, желая только прочесть из него, и другой чувак не сможет получить доступ даже для чтения. Из плюсов — конфликты никогда не случаются, всё можно делать в автоматическом режиме. Минусы — снижается производительность, параллелизм резко падает.
Оптимистическая — кто первый встал, того и тапки. Параллелизм ок, но приходится разрешать конфликты.
Выбираем между ними по следующим критериям:

- частота доступа;
- длительность доступа;
- возможность разрешения конфликтов.

В VCS оптимистические. В каком-нибудь биллинге — пессимистические, ибо некому разрешать конфликы, изменения короткие и их много.
Но! Пусть есть три класса.



Стрелочками обозначены зависимости между ними. Возьмём пессимистическую блокировку: один чувак на A, другой — на B. Не столкнутся по блокировке, но, может быть, ничего работать не будет. А всё потому, что блокировку надо накладывать на все ресурсы, от которых мы зависим.
Есть два типа блокировок: exclusive и shared. Вторая — это когда залочить надо, но менять мы не собираемся, тогда несколько пользователей могут прочитать этот ресурс.
Но даже с этим возможны проблемы, а именно — deadlocks. Они появляются, когда мы, имея shared блокировку, хотим получить exclusive, или «получаем больше блокир.» TODO распарсить. Разрешаем выбором жертвы, которую надо убить: по таймауту или по принципу «ты мне не нравишься».
Впрочем, сам по себе дедлок не страшен — например, в биллинге всё просто распределится на другие потоки. Если возможны дедлоки, то приложение должно иметь это в виду.

7.3 Основные свойства транзакции

- **A**tomicity — транзакция или вся целиком выполняется, или вся целиком не выполняется;
- **C**onsistency — каждая транзакция приводит к валидным данным;
- **I**solation — транзакция не может работать с ресурсами транзакций, которые ещё не завершены;
- **D**urability — результаты завершённых транзакций не будут утеряны.

Эти свойства далеко не везде выполняются.

1. Phantom objects — из-за второй транзакции в нашей появляются новые объекты;
2. Unrepeatable reads — выполняем чтение, в соседней транзакции поменяли, снова выполняем чтение;
3. Garbage reads — видим данные из транзакций, которые ещё не завершились (уточнить).

Транзакции по соблюдению этих правил бывают *посмотреть в курсе БД и распарсить записи в тетрадке.*

8 Сессии

Анонимный пользователь Михаил[15], отключив куки и включив порно-режим, изучает книжки на амазоне. Каждый запрос не зависит от предыдущего из-за параноидальности, это — сессия без состояния (stateless session). Некоторое время назад большая часть была такой. Теперь Михаил ходит по сайту, не залогинившись, и везде есть плашечка «вы недавно посмотрели». Это — statefull session, сессия с состоянием.
И, наконец, он решил что-то купить. Это тоже statefull (но action, а не view, как предыдущие? уточнить).
Как сохранить состояние? Три варианта:

1. самый простой — на стороне клиента (cookies, ...) (client-side);

[15] «Если мы параноики, мы не подключены к интернету»

2. server-side;

3. database.

Если всё на клиенте, для сервера в каждом запросе вся инфа, что ему нужна. Должна быть mutable с точки зрения сервера. Server-side — у клиента session_id, он передаёт его серверу; guid[16] гарантированно уникальный и считается относительно дорого. DB — ???
Что будет, если Михаил вдруг закроет ноут? client-side — ничего, просто сессия заэкспайрится когда-то. server-side — надо будет почистить память (это не persistent storage, так что может и почистить память). DB — точно надо подчистить.
С точки зрения durability в случае отказа база всегда есть, сервер может того, как и клиент. В базе всё ок, что не умерло, всё подняли из базы. А если клиент умер, то надо подчистить его контекст на сервере.

8.1 Кластеризация

У нас есть приложение для просмотра котиков, у нас в 10 раз больше подписчиков, чем у инстаграма, мы в десять раз убыточнее, нас покупают в десять раз дороже. Хотим кластеризоваться. Первый вариант — а-ля сбер: берём самую мощную IBM-железку в мире (таких всего 3 инсталляции), хотим туда что-то от оракла, и оказывается, что это единственная инсталляция и ребята из оракла не знают, что делать. Второй вариант — увеличиваем число серверов, но каждый запрос от клиента проходит через Load Balancer (nginx, apache...).
Рассматриваем, понятное дело, его.
Если всё на клиенте, то проблем нет — всё своё ношу с собой. Иначе варианты:

- sticky session — в рамках одной сессии пользователь всегда на одном и том же сервере. Проблемы: может быть *(кто, что?)* простой одних серверов при перегруженности других, а если сервер сдохнет, сессия может потеряться.

- round-robin??? кто меньше загружен, туда и отправляем. Делается на LB.

Боремся с перегруженностью: session transfer. Сервера всё же взаимосвязаны и умеют обмениваться сессиями. Только надо ещё уведомить LB.
Боремся со смертью сервера: distributed session (но надо реплицировать все сессии, а это дорого), distributed memory cache (получаем информацию о сессии из кэша, которые отделно от серверов).

9 Распределённые вычисления

Люди раньше, начитавшись про корбу и RMI, все объекты (client, address, contract, subscriptions...) клали по разным серверам. Вырос контракт — нарастили ноду. Но на запрос между нодами тратится очень много времени, да ещё и стараемся не вдаваться в детали: хотим данные об адресе — получаем вообще весь адрес и разбираем его у себя локально.
Локализация remote-интерфейсов:

1. фасады;

2. ленивая инициализация объектов, которые пришли удалённо;

3. data transfer object.

Бывают ещё SOAP, json-запросы over-http, абстрактные сериализаторы типа protobuff'a.
Этот раздел был в конце лекции, и какой-то он странный

[16] Globally Unique Identifier

10 Release Management[17]

Как всё обычно делается? «фигак, фигак и в продакшн»[18]
Была у нас общая папка для хранения исходников, получились проблемы, как уже было рассказано ранее, перешли к version control. Многие используют его как ту папку — одна большая толстая ветка. Было CVS — были и бранчи, и теги, а каждый релиз — склеивание бранчей на денёк. Потом был SVN, стало немного гуманнее в этом. Потом появился приличный VC.
Есть два пути: побранчуемся и склеимся, потом в продакшн, или continuous delivery[19].
CD — это полный-полный Agile в худшем понимании этого слова.
[тут, возможно, будут картинки]
Вывод из картинок: бранчи должны быть короткие.
Создавать новые бранчи можно по принципам:

- Branch per Release — был популярен в централизованных VC. Все работают над тем, что идёт в продакшн, но все работают над одним и тем же кодом;
- Branch per Task;
- Promotional Branch (выпустили 1.1, выпустили 1.2, и тут оказалось, что надо пропатчить 1.1, и так и идёт тысяча ужасных ответвлений);
- Branch per Component;
- Branch per OS.

Накладные расходы от бранчевания: надо мёржиться, больше тестирования.
Антипаттерны бранчевания:

- боязнь мёржиться;
- сразу мёржить в мастер (много времени на мёрж, высока вероятность, что код какой-то странный);
- бесконечные бранчи;
- мёрж не в ту сторону (из 1.2 перенесли функциональность в 1.1, потому что какой-то клиент обновляться не хочет);
- отдельные ветки для кучи маленьких изменений (pull request);
- каскадные бранчи (есть бранч на 1.1, от него на 1.2 и так далее, а вообще есть бранч на 2.1...);
- временные бранчи;
- нестабильные бранчи (сбранчевались, пока код не работает);
- бранчи на члена команды.

  «Рано или поздно вы с чем-то из этого столкнётесь».

[17] «Лучше бы про что-то другое поговорил»
[18] Из-за присутствия автора на лекции было сказано именно так.
[19] «Надо потом хорошо стереть, а то скажут, что я вас тут плохому учу». На доске: «Continuous Delivery (не лучший вариант, но, если очень надо, то можно!)»