Конспект по проектированию ПО

v0.2

Содержание

0 Disclaimer

Это конспект по курсу «Проектирование программного обеспечения», прочитанному year2011 в осеннем семестре. 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

[1] Дмитрий Юрьевич Кочелаев

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:

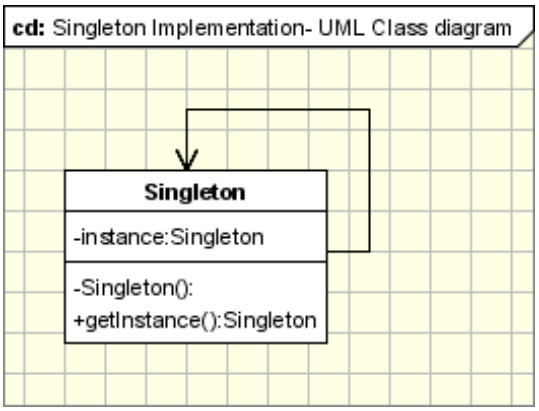
- вычитывание теми, кто был на парах;
- недостающие картинки к паттернам;
- дописать ещё про паттерны;
- убрать пробелы в многослойной архитектуре.

1 Порождающие паттерны[3]

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

1.1 Singleton

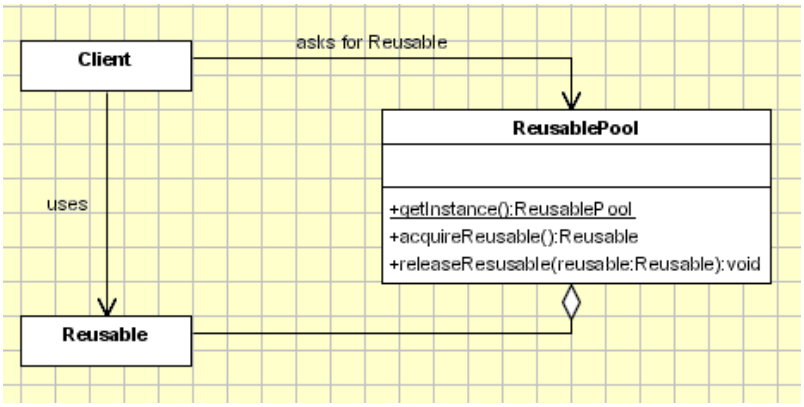
Гарантирует, что у класса есть только один экземпляр, и предоставляет ему глобальную точку доступа.



1.2 Pool[5]

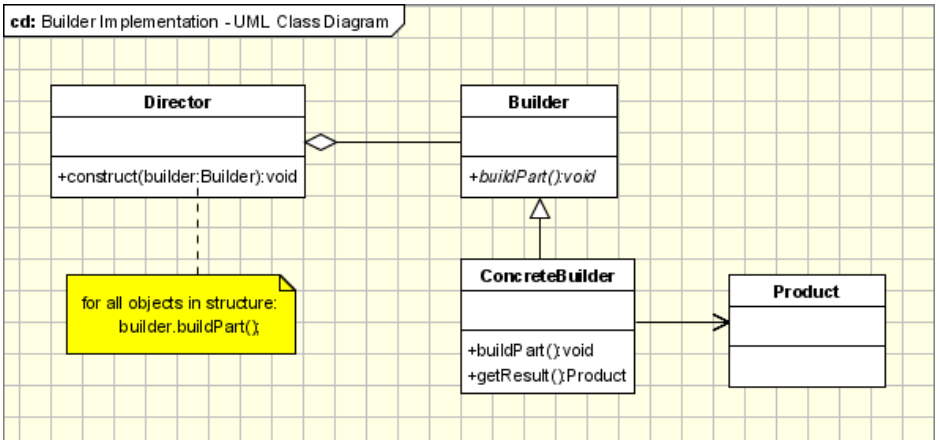
Храним объекты готовыми к использованию: когда нужен, берём из пула (а не создаём), когда не нужен — возвращаем в пул (а не удаляем). Pool объектов никогда нельзя писать самому.

[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”
[5]Нет в GoF.



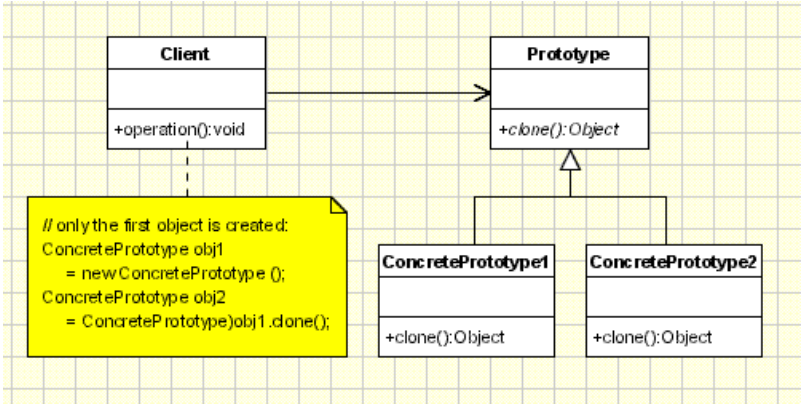
1.3 Builder

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.

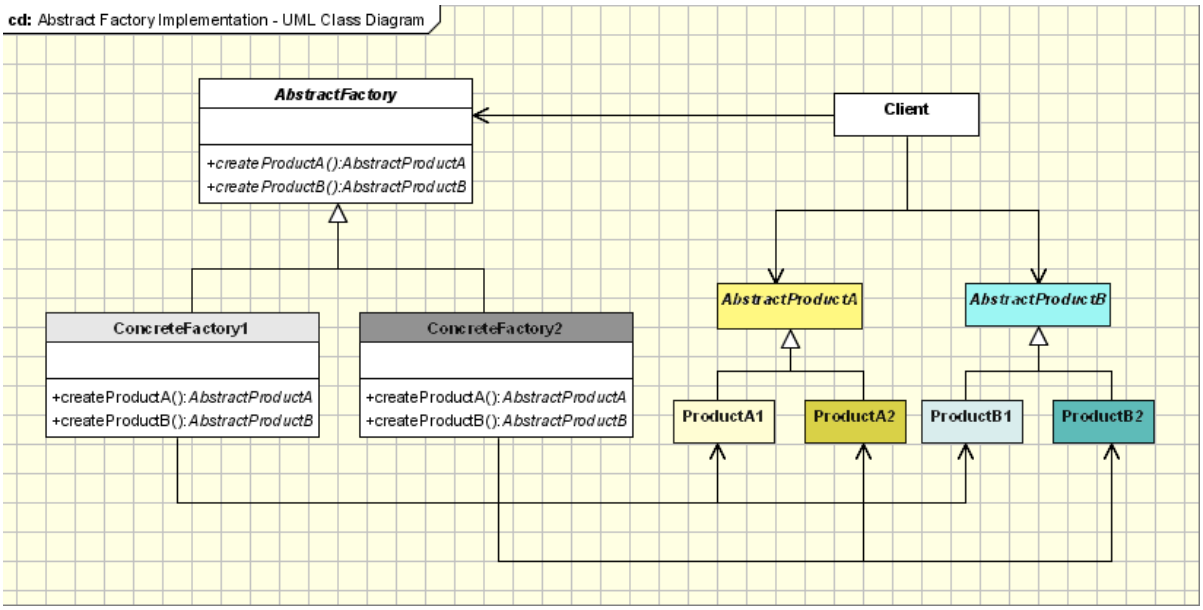


1.5 Factory Method

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

1.6 Abstract Factory

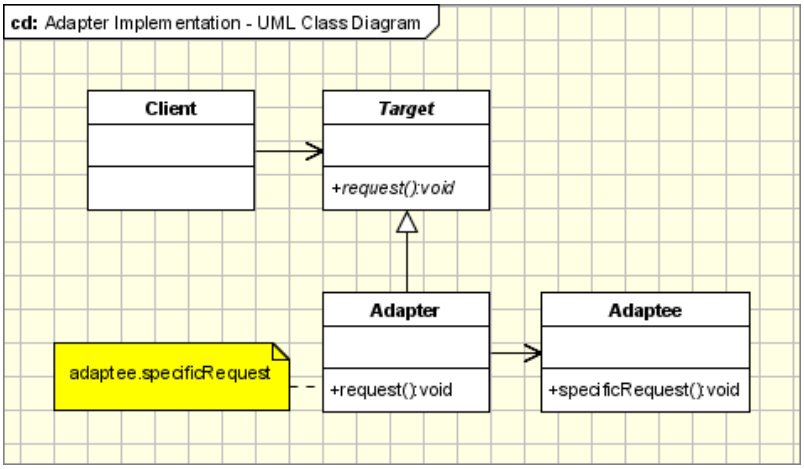
Фабрика фабрик. Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.



2 Структурные паттерны[6]

2.1 Adapter

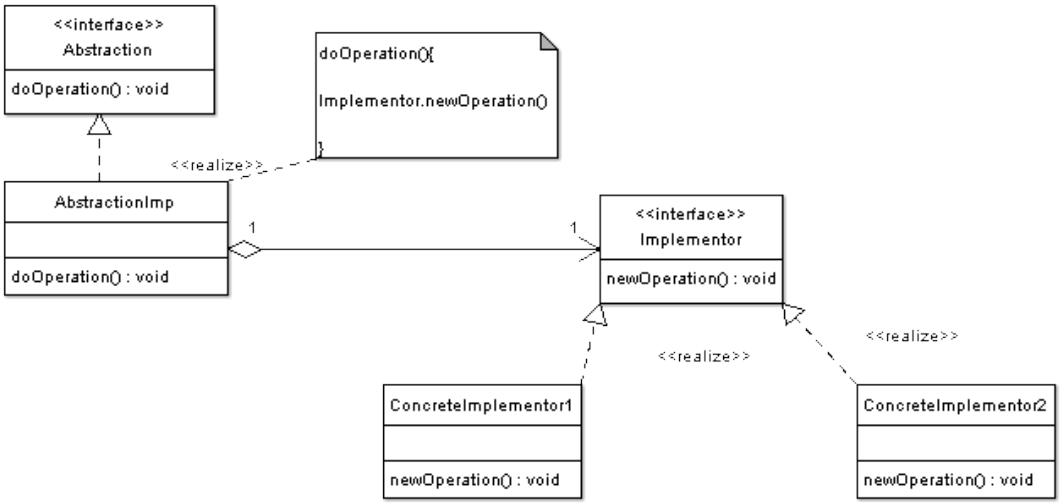
Чтоб скрестить ежа с ужом: для юзания классов, не совместимых по интерфейсу.



[6]И тут тоже.

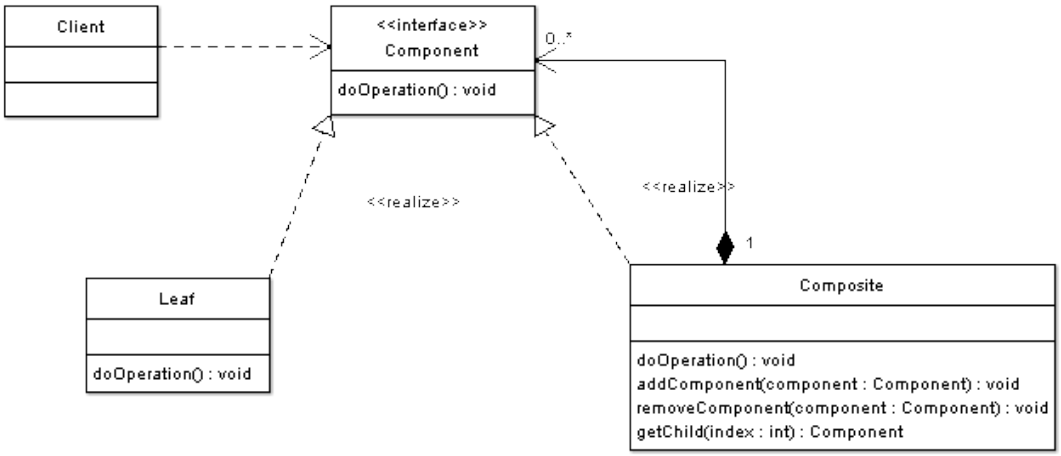
2.2 Bridge

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.

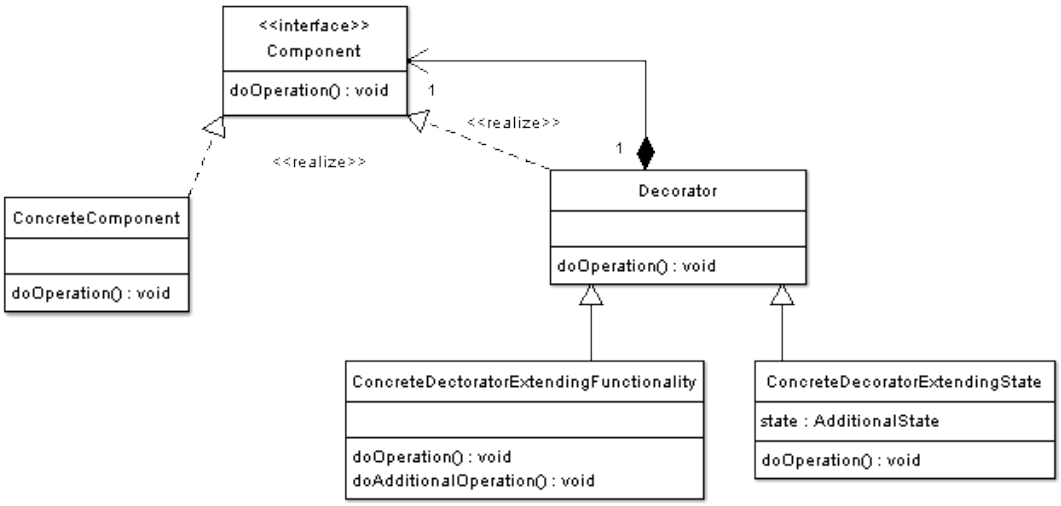


2.4 Facade

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

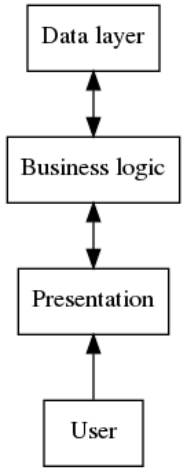


Рис. 1: Классика

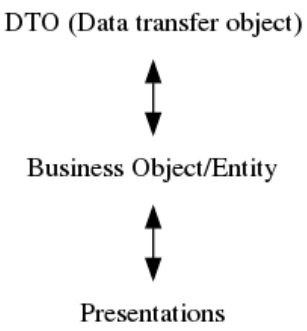


Рис. 2: 1 layer — 1 object

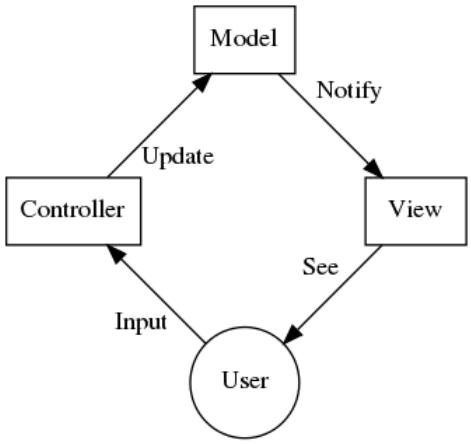


Рис. 3: MVC

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)

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

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:

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.

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

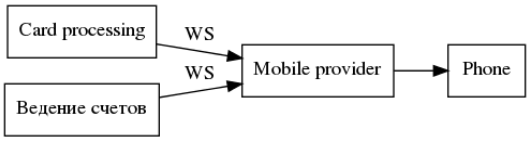


Рис. 4: Плохая схема

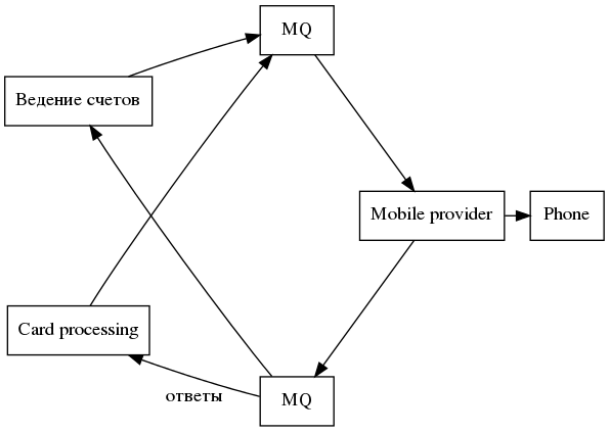


Рис. 5: Хорошая схема

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. 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).

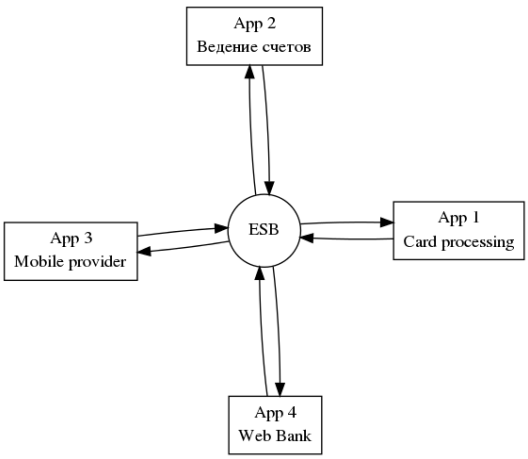


Рис. 6: (Enterprise) Service Bus. Стрелочки чаще всего — очереди

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

[11] «Кто не смотрел “Шестое чувство”: это был спойлер»

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»);
- 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'a, 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:

[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.

[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.

- 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.