# Размышления о принципах проектирования

**ЦИТАТА**: *Век живи — век учись! И ты наконец достигнешь того, что, подобно мудрецу, будешь иметь право сказать, что ничего не знаешь.* Козьма Прутков Или: *Отыщи всему начало, и ты многое поймешь.*

Для чего понадобилось кому-то выдумывать разные паттерны проектирования, принципы и методики разработки? Разве не было бы проще просто научить разработчиков хорошему проектированию? Или почему, тогда уже, не формализовать процесс разработки и не ввести четкие количественные метрики дизайна (\*), которые бы говорили о качестве решения?

(\*) Сноска. Сложности перевода. В русскоязычной литературе нет однозначного перевода термина "software design", который бы точно передавал смысл оригинала. Когда понятие "design" описывает процесс, то его принято переводить как "проектирование". Но все становится сложнее, когда понятие "design" описывает результаты этого процесса. Что мы получаем в результате процесса проектирования? Проект? Дизайн? Первый вариант не подходит, поскольку термин "проект" в области программирования означает проектную деятельность. Второй вариант тоже не идеален, поскольку его легко спутать с дизайном пользовательских интерфейсов. Меньшим злом, с моей точки зрения, является использование разных вариантов перевода для описания разных значений этого термина. Поэтому, в данной книге будет использоваться термин "проектирование" для описание процесса проектирования, и термин "дизайн" для описания артефактов процесса проектирования.

"Правильный дизайн" - это святой Грааль молодых разработчиков и молодых менеджеров. И те, и другие мечтают найти ответ на главный вопрос разработки ПО – как добиться качественного дизайна, в сжатые сроки и с минимумом усилий. Со временем и молодой разработчик, и молодой менеджер приходят к пониманию того, что это невозможно. Невозможно найти идеальный абстрактный дизайн, поскольку слова "идеальный" и "абстрактный" противоречат друг другу. Проектирование – это постоянный поиск компромисса между противоречивыми требованиями: производительностью и читабельностью, простотой и расширяемостью, тестируемостью и цельностью решения. Даже если учитывать, что разработчик всегда решает правильную задачу, а не борется с ветряными мельницами, нельзя "абстрактно" сказать, какие характеристики дизайна являются ключевыми здесь и сейчас.

Существуют формальные критерии, которые описывают качество кода или дизайна: цикломатическая сложность методов, глубина иерархии наследования, число входящих и исходящих связей класса или модуля, число строк метода, в конце концов. Эти количественные показатели полезны, но попадание их в заданные границы является необходимым, но недостаточным условием хорошего дизайна. Если классы разбиты неумело, а важные абстракции предметной области не выявлены, то какими бы количественными характеристиками не обладал дизайн, он никогда не будет хорошим.

Помимо формальных критериев, есть универсальные понятия дизайна – связанность (coupling) и связность (cohesion) (\*). Качественный дизайн обладает слабой связанностью (low coupling), и сильной связностью (high cohesion). Это значит, что программный компонент имеет небольшое число внешних связей, и отвечает за решение близких по смыслу задач. Данные свойства полезны, но слишком неформальны.

(\*) Сноска. Это еще одна пара терминов, перевод которых на русский язык вызывает сложности. Существует несколько популярных вариантов перевода этих понятий, но ни один из них не вызывает четких ассоциаций в голове у читателя. Чтобы устранить эту неоднозначность, при использовании этих понятий в тексте книги, я всегда буду приводить оригинальное значение. **TODO**: тут нужно проверить, возможно, эту сноску нужно дать раньше, а не во вступлении первой части. Скорее всего, эти темрины встречаются и раньше. Тогда сноску нужно будет перенести в место их первого использования!!

Между формальными и неформальными критериями находятся принципы проектирования – набор правил, на которые опираются опытные проектировщики. Цель принципов: простыми словами описать, "что такое хорошо, а что такое плохо" в вопросах проектирования. «Идеальный класс должен иметь лишь одну причину для изменения, обладать минимальным интерфейсом, правильно реализовывать наследование и предотвращать каскадные изменения в коде при изменении требований».

Бертран Мейер в своей книге "Agile!: The Good, The Hype, and The Ugly" (\*) дает достаточное четкое определение того, что такое принцип проектирования:

(\*) Bertrand Meyer, "Agile!: The Good, The Hype, and The Ugly", Springer, 2014

"Принцип – это методологическое правило, которое выражает общий взгляд на разработку ПО. Хороший принцип является одновременно *абстрактным* и *опровергаемым* (falsifiable). **Абстрактность** отличает принцип от практик, а опровергаемость отличает принцип от банальности (platitude). Абстрактность означает, что принцип должен описывать универсальное правило, а не конкретную практику. **Опровергаемость** означает, что у разумного человека должна быть возможность не согласиться с принципом. Если никто в здравом уме не будет оспаривать предложенный принцип, то это правило будет полезным, но не интересным. Чтобы правило являлось принципом – независимо от вашего мнения – вы должны предполагать наличие людей, придерживающихся противоположной точки зрения."

### Использование принципов проектирования

TODO: найти такую же картинку, но с copywritами.



Разработчик за свою профессиональную карьеру проходит несколько стадий владения таким инструментом, как принципы проектирования.

**На первой стадии** молодой разработчик еще не дорос до абстрактных принципов, вместо этого он ищет набор конкретных практик, которые позволят ему получить качественное решение. «Я хочу узнать набор шагов, четкое следование которым приведет меня к нужному результату!». На этом этапе разработчик хорошо копирует чужое решение, и сталкивается с серьезными трудностями, если описание слишком абстрактное или не абсолютно подходит к его случаю.

**На второй стадии** разработчик начинает понимать, что лежит в основе конкретной практики, и для чего нужны принципы проектирования. «Ага, этот класс нарушает принцип единой обязанности, поскольку он ходит в базу и содержит бизнес логику! Получается, что у него есть две четкие причины для изменения!».

Несмотря на возросший уровень, именно на этом этапе происходит наиболее частое использование принципов или паттернов не по назначению. Легко доказать, что конкретный класс нарушает данный принцип, но на этом этапе развития не всегда очевидно, когда это нарушение оправдано, а когда нет.

**На третьей стадии** у разработчика (или уже скорее Архитектора) развивается чутье и появляется довольно четкое понимание, какую проблему призван решить конкретный принцип проектирования. Поскольку, по своему определению, принцип не является однозначным, у опытного разработчика появляется понимание, когда нарушение оправдано, когда с ним можно жить, а когда пришло время браться за исправление дизайна.

При достаточном опыте, принципы начинают использоваться на подсознательном уровне. Разделение ответственностей классов и методов происходит на этапе их разработки, а не во время рецензирования кода. На этом этапе принципы не управляют дизайном приложения, а начинают играть скорее коммуникативную роль для общения с младшими коллегами: «Смотри, у тебя класс делает слишком многое, значит он нарушает принцип единой обязанности! А у этого класса есть два вида клиентов, каждый из которых использует этот класс по-своему, значит он нарушает принцип разделения интерфейсов!».

**ПРИМЕЧАНИЕ** В боевых искусствах принято выделять три стадии мастерства: сю, ха, и ри (Shu, Ha, Ri: https://en.wikipedia.org/wiki/Shuhari). На первой ступени находится ученик, который лишь повторяет движения за мастером. На второй ступени ученик начинает освобождаться от правил и сам начинает решать, когда им следовать, а когда – нет. На третьей стадии правила пропадают, ученик становится мастером и может сам эти правила создавать. Эта же модель, но несколько под иным соусом появилась и в американской культуре, под названием Модель Дрейфуса: http://en.wikipedia.org/wiki/Dreyfus\_model\_of\_skill\_acquisition.

### Правильное использование принципов проектирования

Применение любого принципа проектирования имеет свою цену. Дробление класса на более мелкие составляющие, чтобы он отвечал принципу единственной обязанности, может привести к "размазыванию" логики по нескольким классам. А это может привести к уменьшению связности (low cohesion) и читабельности, а иногда, и к падению производительности.

Нарушение принципа Открыт-закрыт может быть оправдано вопросами обратной совместимости. Мы можем игнорировать принцип замещения Лисков, поскольку наследование не всегда определяет отношение ЯВЛЯЕТСЯ. Интерфейс может быть толстым из-за обратной совместимости или удобства использования. А инверсия зависимостей легко может подорвать инкапсуляцию и привести к чрезмерному числу уровней косвенности.

Чтобы эффективно использовать принципы нужно знать, какую проблему они на самом деле решают и является ли она для вас актуальной. Обычно, важность принципов возрастает с увеличением сложности или при увеличении стоимости внесения изменений. Ключевая бизнес-логика приложения является одним из примеров того, что должно быть изолировано от остального мира. Любые публично доступные классы должны иметь минимум дополнительных связей с внешним миром и должны легко позволять делать простые вещи. Классы, которые находятся на стыке модулей должны быть продуманы лучше остальных.

Ключом для получения хорошего дизайна и эффективного использования принципов проектирования является итеративный подход к проектированию. На ранних этапах мы мало что понимаем в предметной области, поэтому все, что мы можем сделать – это разбить модули по ролям: инфраструктуру отдельно, логику отдельно. При этом автономность классов и минимизация побочных эффектов делает решение проще. Когда класс обладает минимальным числом зависимостей, то его влияние на систему, и влияние системы на него, строго ограничено.

По мере развития приложения улучшается понимание предметной области. Одновременно с этим, приложение обрастает заплатками и неуклюжими решениями. Накапливается технический долг, который приводит к усложнению добавления новых возможностей, а исправление одной ошибки приводит к появлению двух других. Чтобы не доводить систему до такого состояния, лучше постоянно думать о дизайне и улучшать его по ходу развития. Перед реализацией новой возможности или исправлением ошибки, я стараюсь ответить на такой вопрос: отвечает ли текущий дизайн моему новому пониманию задачи, или требуется его переработка?

Главной движущей силой изменений, в конечном счете, является возрастающая сложность. Как только мне не удается держать в голове текущее решение, или простое изменение требований приводит к изменению целой группы классов, то значит пришло время подумать над изменением дизайна. "Класс стал слишком сложным, он нарушает принцип единой обязанности, пришло время разбить его на два. Информация об иерархии наследования распространилась по всему модулю, наверное, стоит перенести часть логики в базовый класс, а саму иерархию спрятать за фабрикой. Число зависимостей класса стало слишком большим, значит пришло время пересмотреть обязанности класса или объединить низкоуровневые зависимости."

При этом я всегда проверяю, не приведут ли меня принципы и паттерны в мир ненужной сложности (overengineering): стал ли мой дизайн после внесения изменений проще? Не решаю ли я проблему, которой на самом деле не существует? Не попал ли я в сети преждевременного обобщения (premature generalization)? Иногда приходится провести несколько итераций, прежде чем удается найти разумное решение задачи.

### Антипринципы проектирования

Популярность принципов проектирования легко может сыграть с вами и вашей командой злую шутку. Чрезмерная любовь к принципам и паттернам может проявляться в виде оверинжиниринга и чрезмерной сложности. Но зная об этом, мы можем подготовиться и предугадать, как именно будет проявляться чрезмерная любовь принципов в коде приложения.

**Anti-SRP – Принцип размытой обязанности**. Классы разбиты на множество мелких классов, в результате чего логика размазывается по нескольким классам/модулям.

**Anti-OCP – Принцип фабрики-фабрик**. Дизайн является слишком обобщенным и расширябельным, выделяется слишком большое число уровней абстракции.

**Anti-LCP – Принцип непонятного наследования**. Принцип проявляется либо в чрезмерном количестве наследования, либо в его полном отсутствии, в зависимости от опыта и взглядов местного главного архитектора.

**Anti-ISP – Принцип тысячи интерфейсов**. Интерфейсы классов разбиваются на слишком большое число составляющих, что делает их неудобными для использования всеми клиентами.

**Anti-DIP – Принцип инверсии сознания или DI-головного мозга**. Интерфейсы выделяются для каждого класса и пачками передаются через конструкторы. Понять, где находится логика становится практически невозможно.