Антипаттерны

То, как делать не надо

Юрий Литвинов
yurii.litvinov@gmail.com

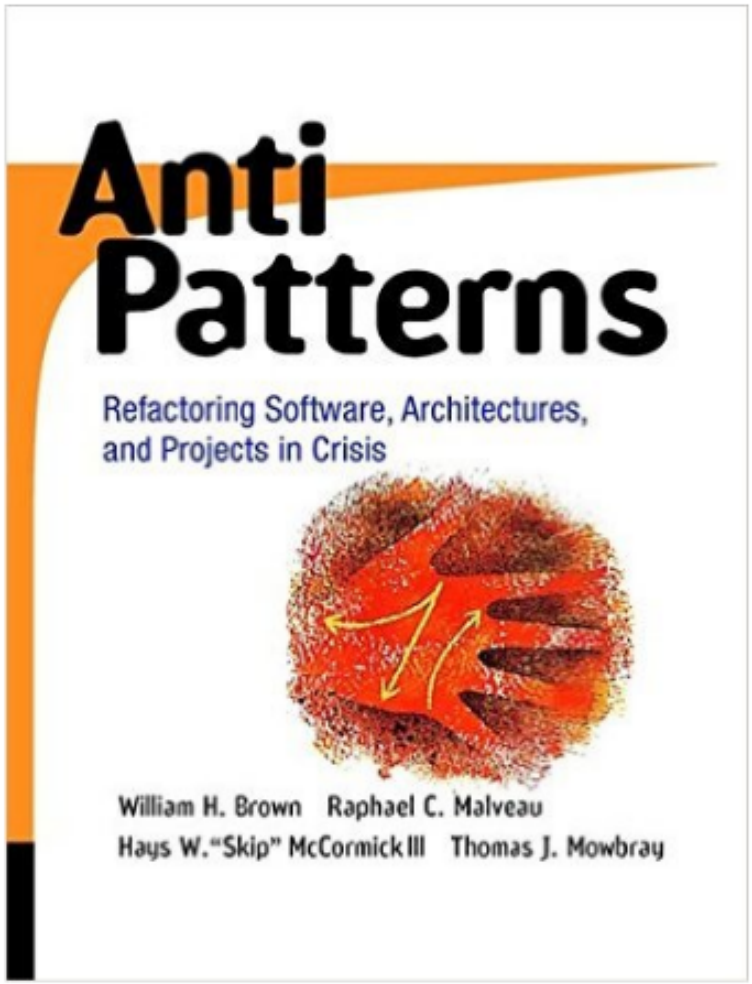
10.05.2017г

Антипаттерны — что это и зачем?

- Часто встречающиеся решения, приводящие к известным проблемам
  - Сами по себе решения могут быть неплохи, может быть плох контекст их применения
- Так же, как и паттерны, нужны для введения общего словаря и общеизвестного набора решений
- Описание антипаттерна должно содержать не только проблему, но и то, почему это решение плохо, и как сделать хорошо
- Бывают разные виды, относящиеся к разным сферам деятельности
  - Антипаттерны реализации
    - В том числе, специфичные для конкретного языка или технологии
  - Архитектурные антипаттерны
  - Антипаттерны организации

Книжка

AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis by William J. Brown, Raphael C. Malveau, SkipMcCormick, Thomas J. Mowbray, Wiley, 1998, 336pp.



Семь причин провала проектов

- Спешка
  - Look you — just “clean up” the code. We ship tomorrow.
- Апатия — неприменение известных хороших решений
  - Reuse? Who’s ever gonna reuse this crappy code? NO ONE! That’s who.
- Недалёкость — незнание хороших решений
  - I don’t need to know and I don’t care to know.
- Лень — следование пути наименьшего сопротивления
- Архитектурная жадность — излишняя детальность
  - Well, it certainly is complicated! I’m sure our clients will be very, very impressed.
- Неведение — нежелание понимать
- Гордость — нежелание переиспользовать готовые решения
  - Not-invented-here syndrome.

Circular dependency

- Два или больше компонентов, которые зависят друг от друга
  - Зависимости модулей должны образовывать ациклический граф
- Часто появляется как результат попыток добавить callback-и
  - Или просто по неосторожности
- Имеет впечатляющие последствия в C++, во многих языках невозможен
- Как бороться:
  - Зависеть от абстракции, а не от реализации
  - Разделение на слои
  - Observer, Dependency Injection и т.д.

Sequential coupling

- Необходимость вызывать методы класса в определённом порядке
  - Например, вызов init() после конструктора
  - Бывают более запущенные случаи, когда есть целая цепочка методов
- Как бороться:
  - Шаблонный метод
  - Фабрики и строители

Call super

- Необходимость вызывать из переопределённого метода потомка переопределяемый метод предка
  - Не может быть проверено компилятором
- Как бороться:
  - Template Method
    - **Позволяет** переопределить поведение предка, а не **требует** этого

Yo-yo problem

- Развитие идеи Call Super — а давайте предок будет тоже вызывать виртуальные методы, переопределённые в потомке, которые будут вызывать методы предка и т.д.
  - Красивая объектно-ориентированная архитектура же!
- Как бороться:
  - Перераспределить функциональность между предками и потомками
  - Возможно, распилить иерархию на несколько
    - Паттерн “Мост”
  - Вообще избегать глубоких иерархий наследования
    - И ещё более вообще, использовать наследование только для полиморфных вызовов

Busy waiting

- Ожидание наступления некоторого события в бесконечном цикле с неблокирующими вызовами проверки наступления события
  - Ещё использование циклов для задержек
- Не всегда плохо, может быть валидным решением на встроенных устройствах
- Как бороться:
  - Использовать планировщик: блокирующие вызовы, “усыпляющие” поток до наступления события
    - select в linux
    - Мьютексы, condition_variable и т.д.
  - Использовать аппаратные возможности: таймеры и прерывания

Error hiding

- Сообщение об ошибке прячется за “дружественным к пользователю” сообщением или прячется вообще
  - В худшем случае информация об ошибке теряется окончательно
- Как бороться:
  - Давать программе упасть (ещё, принцип “fail fast”)
  - Логировать все исключения

Magic numbers, Magic strings

- Внезапно появляющиеся в коде программы строковые или числовые литералы
  - 0, 1 и т.д. не считаются
- Особенно забавно, если автор любезно посчитал значение математического выражения и записал в код результат
- Как бороться:
  - Константы
  - Ресурсы для локализации

God Object (The Blob)

“This is the class that is really the heart of our architecture.”

- Один класс управляет всем процессом вычислений, остальные в основном предоставляют ему данные
  - Привычка к структурному программированию, разделение данных и кода
  - Постепенная эволюция proof-of-concept без рефакторинга (лень, спешка)
  - Архитектурная ошибка, неправильное разделение обязанностей
- Хаотичное объединение различных ответственностей в один класс
  - Больше 60 полей и методов могут указывать на God Object
    - При этом String вряд ли так можно классифицировать
- Исключения: обёртки над legacy-компонентами
  - Их нет нужды декомпозировать

God Object, что делать

- Передать больше ответственности классам-данным
- Разделить методы класса на группы, соответствующие контрактам, выполняемым God Object-ом
- Поискать среди уже существующих классов более подходящие для каждой группы методов
- При необходимости создать новые классы, в соответствии с принципом единственности ответственности
- Убрать непрямые зависимости (если объекты A и B лежат внутри объекта G, может так случиться, что A может быть в B, а B — в G)

Lava Flow

“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?!”

- Мёртвый код и незакрытый технический долг “застывают” в системе, как потоки лавы
  - Появляется он, как правило, как эксперименты, “костыли” или быстрые фиксы
  - Люди, писавшие их, уходят из команды, оставшиеся не имеют идей, что это, но трогать боятся (спешка, лень)
    - “It doesn’t really cause any harm, and might actually be critical, and we just don’t have time to mess with it.”
- Закомментированный код, куча TODO, большие методы без комментариев, непонятные интерфейсы

Lava Flow, что делать

- Как предупредить:
  - Не писать production-код до продумывания архитектуры
  - Активно использовать контроль версий с ветками
- Как лечить:
  - Остановить разработку и провести архитектурный реинжиниринг
    - Сложный и долгий процесс анализа существующей системы и создания to-be-архитектуры
  - Постепенное вырезание мёртвого кода приведёт к багам, их нельзя фиксить новыми костылями
  - Выкинуть и написать заново?

Functional Decomposition

“This is our ‘main’ routine, here in the class called LISTENER.”

- Программирование путём разделения задачи на вызывающие друг друга функции
  - Не анти-паттерн для структурного программирования (Pascal, Ada, C) и, тем более, функционального программирования (хотя тут возможны варианты)
    - Высокоуровневая декомпозиция модулей системы по функциям тоже ок
- Классы с “функциональными” именами, классы с одним методом, классы с кучей private-методов

Functional Decomposition, что делать

- Вернуться к требованиям, построить модель предметной области
- Отобразить требования и модель на существующий код
  - Цель — объяснить и задокументировать то, что уже написано
- Классы с одним методом прибить, переместив их код в другие классы
  - Хелперы, например
- Собрать несколько классов в один, который бы отвечал за что-то разумное из предметной области или требований
- Если в классе нет состояния, имеет смысл сделать его функцией или статическим классом
  - Состояние имеет свойство заводиться само по мере рефакторинга
- Сделать из основного метода God Object, а дальше понятно, что делать

Poltergeists

“I’m not exactly sure what this class does, but it sure is important!”

- Классы, которые не нужны архитектуре системы и существуют лишь чтобы выполнить какие-то действия с другими классами
  - ...Controller, ...Manager и т.д., нужные только для инициализации или вызова других классов
  - Время жизни их объектов ограничено (отсюда название)
  - Тенденция использовать побочные эффекты
    - Вся идиома RAII в C++

Golden Hammer

“I have a hammer and everything else is a nail.”, “Our database is our architecture.”, “Maybe we shouldn’t have used Excel macros for this job after all.”

- Рост опыта во владении конкретным продуктом, технологией или стеком пропорционален желанию использовать их везде, где только можно
  - Усугубляется желанием вендоров выпускать кучу расширений
  - Иногда это осмысленно, чаще нет
    - Пересесть на новый язык программирования и стек технологий занимает не больше недели!
  - Собственно, антипаттерн — даже не хотеть погуглить
- Создаваемые решения испытывают сильное влияние конкретной технологии, используемой для их разработки
- Большие вложения в молоток, которые жалко терять

Golden Hammer, как бороться

- Психологические аспекты, которых мы здесь не касаемся
- Обучение
  - Внутрикорпоративные семинары
  - Поездки на конференции
- Разделение системы на заменяемые компоненты
  - Сервисно-ориентированная (микросервисная) архитектура
  - Следование индустриальным стандартам при определении границ компонентов
  - Использование “кросс-технологических” инструментов (Protobuf, Thrift, ...)
- Не бояться нового
  - Квалифицированный программист может хорошо программировать на чём угодно

Design smells

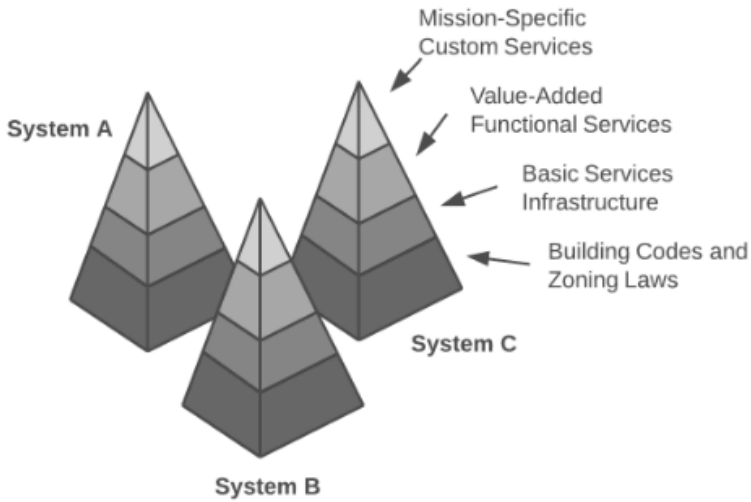
- **Missing abstraction** — использование примитивных или слишком общих типов данных вместо своих классов
- **Multifaceted abstraction** — нарушение единственности ответственности
- **Duplicate abstraction** — две абстракции имеют одно предназначение
- **Deficient encapsulation** — слишком широкая видимость
- **Unexploited encapsulation** — явные проверки типов вместо полиморфизма

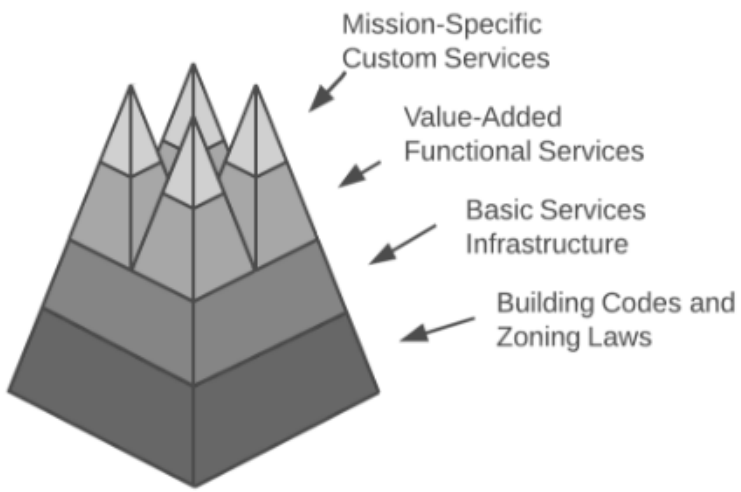
Design smells (2)

- **Broken modularization** — неуместное разбиение на классы/пакеты
- **Insufficient modularization** — неуместное неразбиение на классы/пакеты
- **Cyclically-dependent modularization** — циклические зависимости
- **Unfactored hierarchy** — дупликация в иерархии
- **Broken hierarchy** — нарушение принципа подстановки и IS-A
- **Cyclic hierarchy** — зависимость от потомков

Stovepipe Enterprise (Island of Automation)

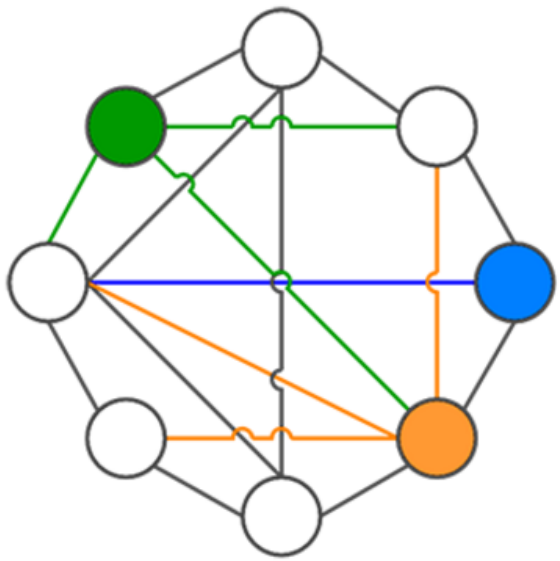
- Отсутствие переиспользования между системами в организации или между компонентами в одной системе
  - Существующие кодовые базы никак не помогают друг другу
- Как бороться:
  - Использовать промышленные стандарты
  - Фиксировать технологии и типовую архитектуру на уровне предприятия
  - Создать и поддерживать инфраструктуру переиспользования





Stovepipe System

- Аналог Stovepipe Enterprise для отдельной системы
  - Отсутствие единого стандарта по взаимодействию между подсистемами
  - Интеграция между подсистемами по принципу ad-hoc
- Как бороться:
  - Абстракция — единые интерфейсы для подсистем
  - Единый протокол общения между подсистемами
    - Паттерны интеграции, например, Enterprise Service Bus



Vendor Lock–In

- Жёсткая зависимость архитектуры или реализации от третьестороннего коммерческого решения
- Приложения могут жить очень долго и легко переживают вендоров третьесторонних компонент или инструментов
- Как бороться:
  - Программировать не “на”, а “с помощью”
  - Изоляционный слой
  - Реинжиниринг

Architecture By Implication

- Отсутствие явных архитектурных спецификаций для разрабатываемой системы
  - “We’ve done systems like this before!”, “There is no risk; we know what we’re doing!”
- Скрытые риски, возможное недопонимание, “Золотой молоток”
- Как бороться:
  - Фиксировать формат описания архитектуры системы
    - Design Document
    - Десятки разных архитектурных фреймворков

Design By Committee

- Попытка принимать архитектурные решения большинством голосов
  - “A camel is a horse designed by a committee.”
- Попытка включить в дизайн идеи, соображения и индивидуальные предпочтения каждого приводит к сложной, объёмной и внутренне противоречивой архитектуре
- Как бороться:
  - Строгий регламент митингов
  - Распределение ролей в команде: product owner, архитектор, разработчики, эксперты и т.д.
  - Идеальный размер команды – 4 человека
    - Тактика, описанная у Брукса, когда программирует один человек, остальные занимаются вспомогательными задачами