2. Проектирование ПО

2.4. Об архитектуре и паттернах проектирования

Признаки удачно спроектированной архитектуры

Эффективность системы

Надёжность
Безопасность
Производительность
Масштабируемость
Отзывчивость
…

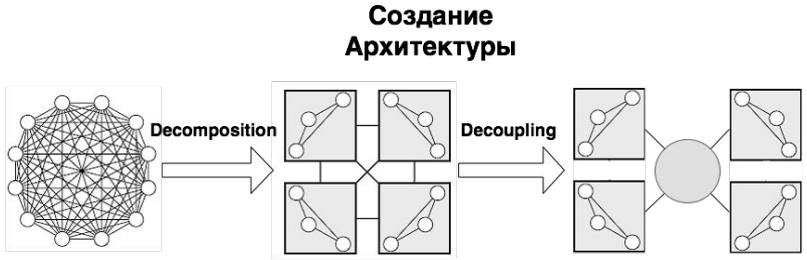
Гибкость системы

Легкость изменения текущей ф-ти
Исправление ошибок
Настройка системы
- Под пользователя
- Под различные сценарии

…

Поддерживаемость системы

Масштабируемость процесса разработки
Тестируемость
Обновляемость
Переиспользуемость
Обратная совместимость
…



Признаки неудачно спроектированной архитектуры

Жесткость

Тяжело модифицировать
Модификация одного модуля влечёт за собой (избыточные) модификации в других

…

Хрупкость

Легкость изменения текущей ф-ти
Изменение в одном модуле нарушают другие модули

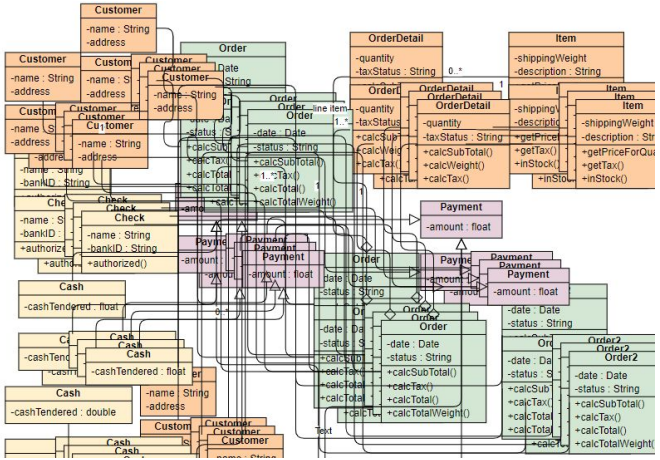
…

Неподвижность

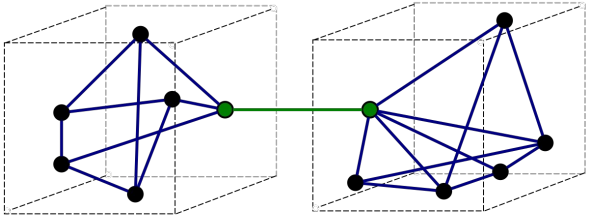
Тяжело “извлечь” модуль наружу

- Как правило, говорит о большом сопряжении и низкой связности некоторых модулей

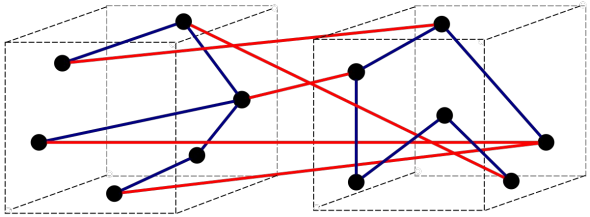
…



Модульность



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

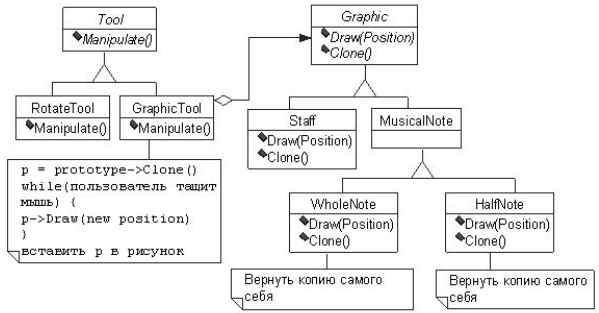
- Метрики взаимозависимости модулей
  - **Coupling (сопряжение)** -- мера того, насколько взаимозависимы разные модули в программе (по вызовам, как правило)
  - **Cohesion (связность)** -- степень, в которой задачи, выполняемые одним модулем, связаны друг с другом (в плане смысла)

Цель -- небольшое сопряжение и сильная связность (low coupling, high cohesion)

Паттерны

- Паттерн (от англ. Pattern) — образец, шаблон.
- В проектировании программ (и не только) -- это разумный, устоявшийся способ решения какой-либо задачи, который точно приведёт к намеченному результату*

- Паттерны ООП -- совсем тактические приёмы
- Архитектурные паттерны -- приёмы проектирования одной или нескольких подсистем
- Архитектурные стили -- что-то совсем глобальное …



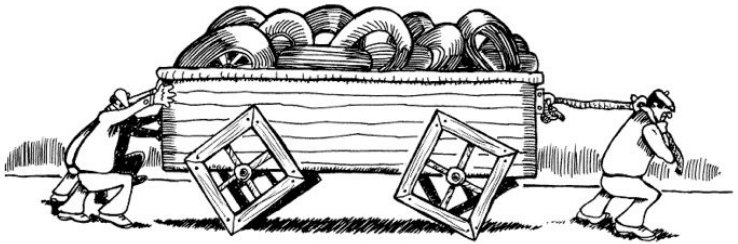
- “Тропа умнее человека” (народное творчество)

Анти-паттерны

- В проектировании программ (и не только) -- это пример того, как не нужно решать те или иные задачи

- В ООП
- В кодировании
- Архитектурные
- Методологические

...



Паттерны проектирования (в ООП)

- В проектировании программ (и не только) -- это разумный, устоявшийся способ решения какой-либо задачи, который точно приведёт к намеченному результату*
- Это некоторая формализация (именованная), которая описывает одну тактическую проблему, способ решения и результат

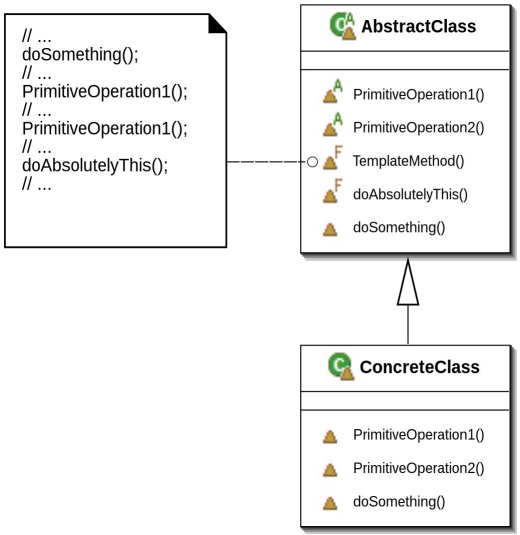
Книга GOF Patterns:
Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Паттерны объектно-ориентированного проектирования / Пер. с англ.: А. Слинкин. — СПб.: Питер, 2021. — 448 с. — ISBN 978-5-4461-1595-2.

Имя	**Задача**
Решение	**Результат**

- “Тропа умнее человека” (народное творчество)

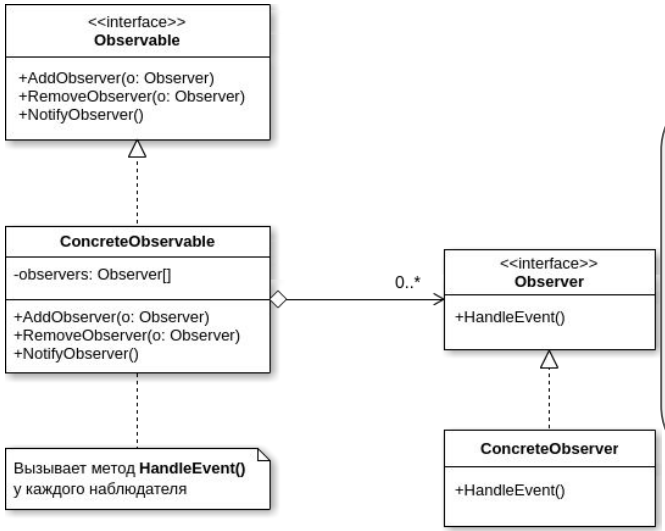
Шаблонный метод (Template method)

- Задача: Реализовать “шаблон” выполнения алгоритма в виде метода
- Когда:
  - Класс является интерфейсом к группе объектов, выполняющих примерно одно и то же (и в том же порядке), но немного по-разному;



Наблюдатель (Observer)

- Задача: реализовать зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически;
- Когда:
  - Когда нужно реализовать “подписку” у группы объектов (1 или более) на события, происходящие в некотором объекте
    - При этом необходимо избежать сильного сопряжения



- **Observable** — интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей;
- **Observer** — интерфейс, с помощью которого наблюдатель получает оповещение;
- **ConcreteObservable** — конкретный класс, который реализует интерфейс Observable;
- **ConcreteObserver** — конкретный класс, который реализует интерфейс Observer.

Структурные паттерны

1/ Адаптер

2/ Декоратор

3/ Фасад

4/ Прокси

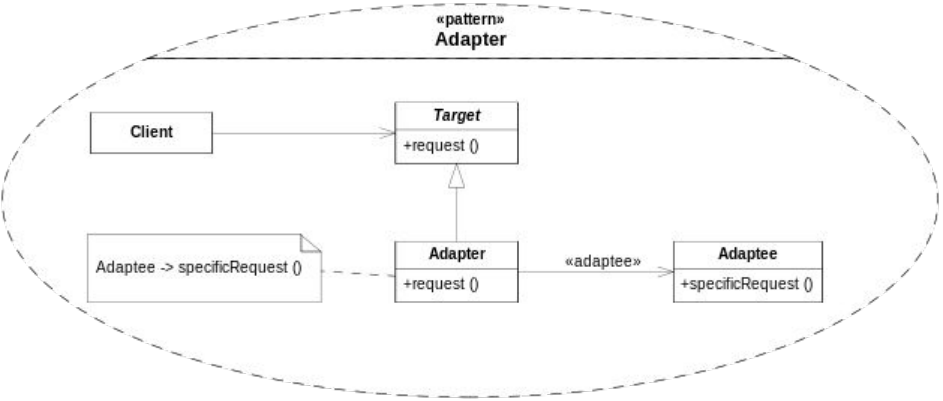
5/ Компоновщик

6/ Приспособленец

7/ Мост

Адаптер (Adapter)

- Задача: позволить объектам с несовместимыми интерфейсами работать вместе
- Когда:
  - система поддерживает требуемые данные и поведение, но имеет неподходящий интерфейс.
  - для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс

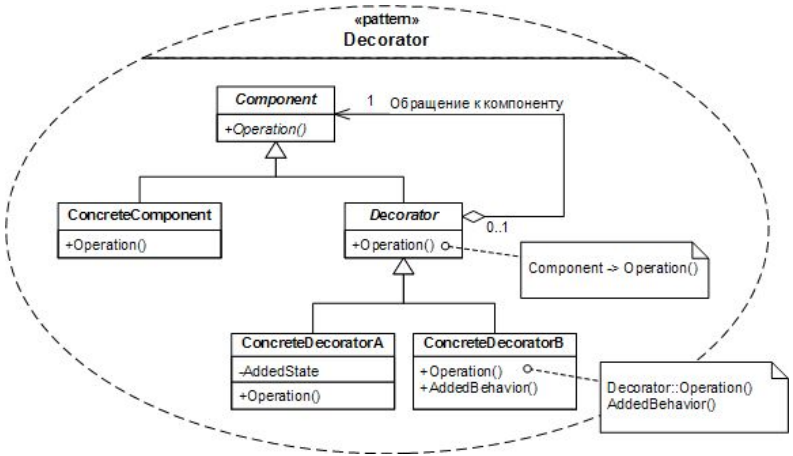


\+ инкапсуляция реализации внешних классов (компонентов, библиотек), система становится независимой от интерфейса внешних классов;

\+ переход на использование других внешних классов не требует переделки самой системы, достаточно реализовать один класс `Adapter`.

Декоратор (Decorator)

- Задача: Подключить дополнительное поведение к объекту
- Когда:
  - Требуется добавить объекту некоторую дополнительную функциональность, которая будет выполняться до, после или даже вместо основной функциональности объекта
  - Но при этом не хочется наследоваться



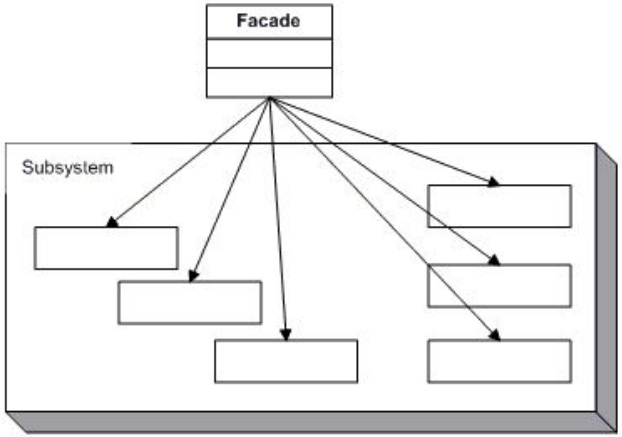
\+ Обеспечивает динамическое добавление функциональности до или после основной функциональности конкретного объекта

\+ Позволяет избежать перегрузки неинтерфейсными классами на верхних уровнях иерархии

\- Оборачивает ровно тот же интерфейс, что предназначен для внешнего мира, что вызывает смешение публичного интерфейса и интерфейса кастомизации, которое не всегда желательно.

Фасад (Fasade)

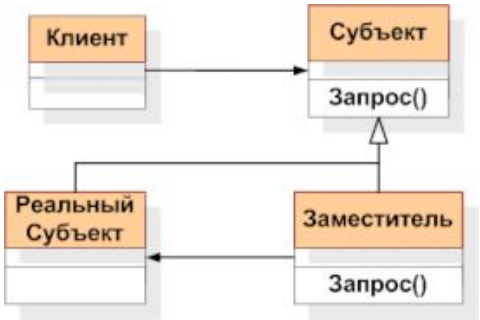
- Задача: обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов для связи с некоторой подсистемой
- Когда:
  - требуется скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам подсистемы
  - нежелательно сильное связывание с этой подсистемой или реализация подсистемы может измениться



\+ Фасад — это внешний объект, обеспечивающий единственную точку входа для служб подсистемы.
\+ Реализация других компонентов подсистемы закрыта и не видна внешним компонентам.
\- Иногда не подходит в виду врожд ённой непрозрачности (см. паттерн “Прокси”).

Прокси, заместитель (Proxy)

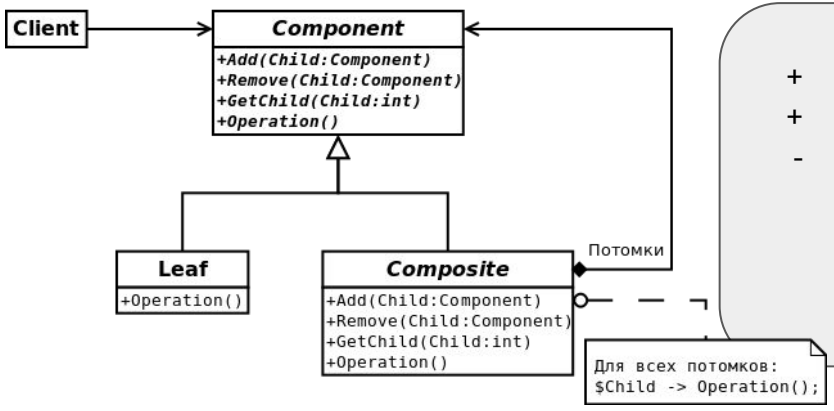
- Задача: предоставить объект, который контролирует доступ к другому объекту, перехватывая все вызовы
- Когда:
  - необходимо контролировать доступ к объекту, не изменяя при этом поведение клиента.
  - необходимо иметь доступ к объекту так, чтобы не создавать реальные объекты непосредственно, а через другой объект, который может иметь дополнительную функциональность.



\+ может выполнять оптимизацию, различные проверки;
\+ Может обеспечить создание реального «Субъекта» только тогда, когда он действительно понадобится
\+ защищает «Субъект» от опасных клиентов (или наоборот);
\- Может ухудшать производительность (доп. вызов и т.д.)

Компоновщик, композит (Composite)

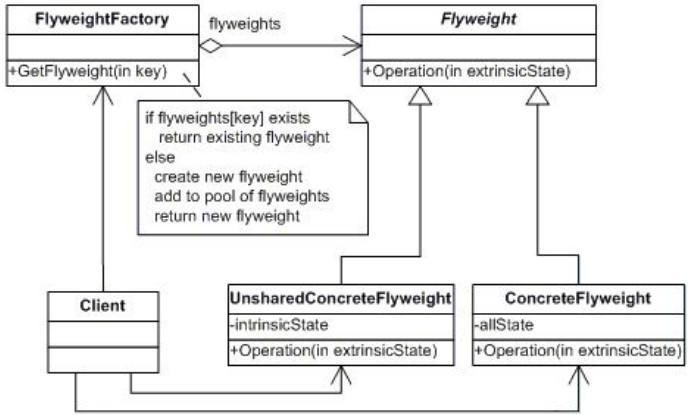
- Задача: организовать объекты в древовидную структуру для представления иерархии от частного к целому
- Когда:
  - необходимо предоставить иерархию по включению.
  - собрать иерархию классов, которые одновременно могут состоять из примитивных и сложных объектов.



\+ упрощает архитектуру клиента;
\+ упростить процесс добавления новых видов объекта
\- возможна избыточная реализация методов, которые отдельные вершины не поддерживают (как минимум, нужно вставить стабики в композит, которые будут либо печатать, что операция не поддерживается, либо кидать исключение, либо запретить их вызов из клиента)

Приспособленец (Flyweight)

- Задача: представить объект как уникальный экземпляр в разных местах программы
- Когда:
  - Оптимизация работы с памятью путём предотвращения создания экземпляров элементов, имеющих общую сущность.



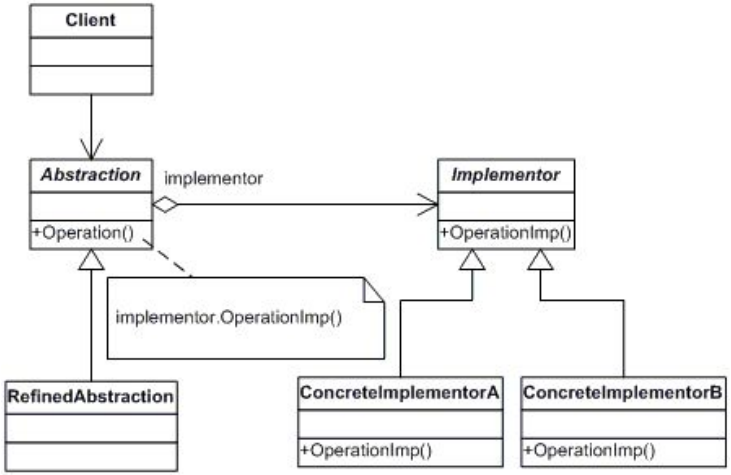
\+ Оптимизирует приложение по памяти

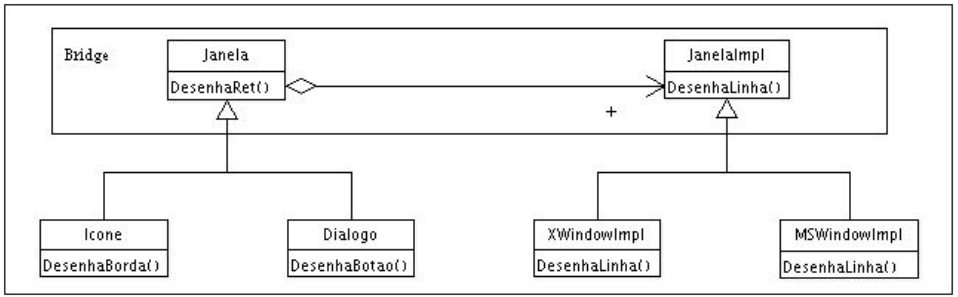
\+ дополняет шаблон “Фабричный метод” таким образом, что при обращении клиента к Factory Method для создания нового объекта ищет уже созданный объект с такими же параметрами, что и у требуемого, и возвращает его клиенту

\-

Мост (Bridge)

- Задача: Разделить абстракцию и реализацию так, чтобы они могли бы модифицироваться независимо
- Когда:
  - Часто меняется не только сам класс, но и то, что он делает
  - Нужно избежать “скачков” вызовов между уровнями иерархии (“Йо-йо” антипаттерн)





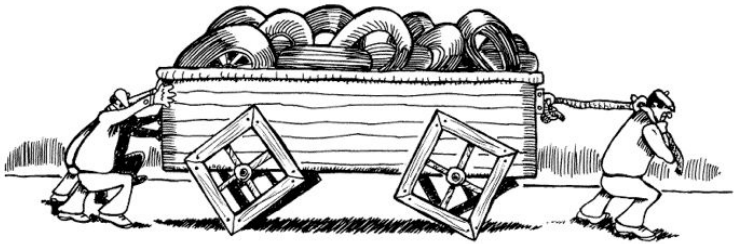
Анти-паттерны

- В проектировании программ (и не только) -- это пример того, как (в общем случае) не нужно решать те или иные задачи

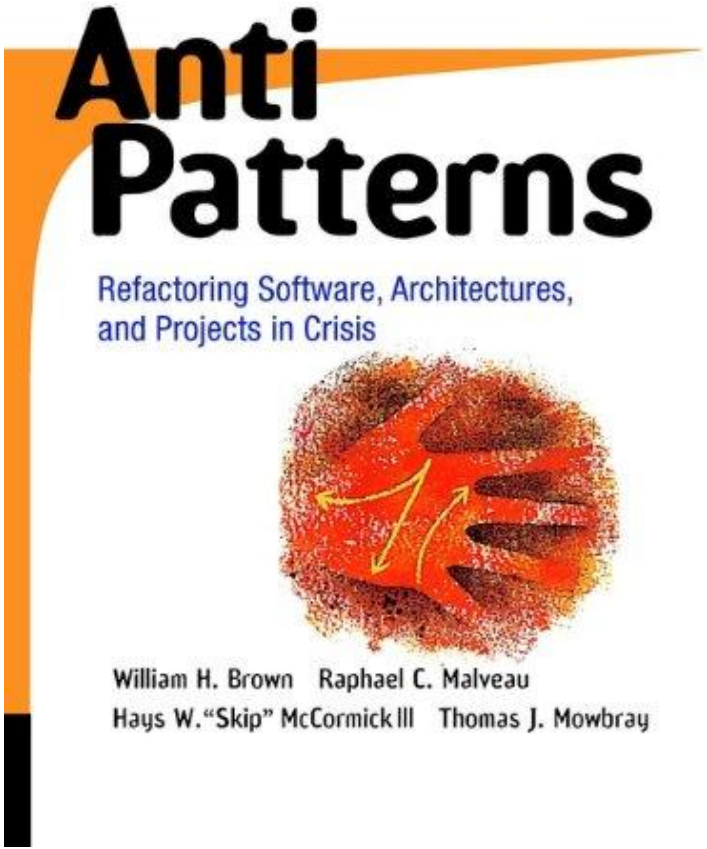
- В ООП
- В кодировании

} Паттерны реализации

- Архитектурные
- Методологические
- ...



Литература


Anti
Patterns
Refactoring Software, Architectures,
and Projects in Crisis
William H. Brown Raphael C. Malveau
Hays W. "Skip" McCormick III Thomas J. Mowbray

Анти-паттерны

7 “грехов”, приводящих к возникновению обрастанию проекта анти-паттернами *

- **Спешка**
  - У меня завтра релиз -- оставлю всё как есть, а позже исправлю…А позже -- ..
- **Апатия**
  - Зачем, если бизнесу, по сути, на это начихать?
- **Недалёкость**
  - Не знаю, и не хочу узнавать. Сделаю всё сам.
- **Лень**
  - Следование пути наименьшего сопротивления
- **Архитектурная жадность**
  - Архитектура настолько подробна, что идеям разработчика не осталось места…
- **Неведение**
  - Нежелание понимать, почему в уже используемом коде содержатся анти-паттерны. Желание переписать всё, не разбираясь.
- **Гордость**
  - Синдром “предубеждения к стороннему коду”

* а более стратегически -- к провалу проекта

Циклическая зависимость

- **Два или более компонента зависят друг от друга**
  - Нарушена ацикличность графа зависимостей
- **Часто проявляется как результат попыток добавить обратный вызов, либо просто по неосторожности**
- **Имеет чудесные последствия в C++, а в ряде ЯП (F# и др) в принципе невозможен**
- **Как бороться:**
  - Зависимость от абстракции, а не от реализации
  - Разделение на слои
  - Паттерн Observer, и др.

```
// file: b.h
class A {
  B _b;
};
// file: a.h
class B {
  A _a;
};
// file main.cc
#include "a.h"
#include "b.h"
int main() {
  A a;
}
```

```
In file included from main.cpp:2:
a.h:3:3: error: ‘B’ does not name a type
    3 |   B _b;
      |   ^
```

Sequential coupling

- **Необходимость вызывать методы класса в определенном порядке**
  - Например, init после конструктора
  - Либо целая цепочка методов, которые, в том числе, выставляют инварианты класса
- **Как бороться:**
  - Шаблонный метод
  - Фабрики (и стратегии, если более обобщённо), строители

Call super

- **Вызов из переопределенного метода потомка переопределяемый метод родителя**
  - **Не может быть проверено компилятором, т.к. связывание позднее**
- **Как бороться:**
  - **Шаблонный метод**
    - **Позволяет переопределить поведение предка, а не требует этого**

Yo-yo problem

- **Транзитивное обобщение call super для многоуровневой иерархии**
  - Давайте родитель также будет вызывать виртуальные методы, переопределенные в потомке, которые будут вызывать методы предка, и т.д.
- **Как бороться:**
  - **Перераспределение функциональности между предками и потомками**
  - **Разделение иерархии на несколько**
    - Паттерн “Мост”
  - **Избегать глубоких иерархий наследования**
    - И вообще, использовать наследование только для полиморфных вызовов

Busy wait

- **Активный цикл ожидания некоторого события, использование циклов для задержек**
  - Не является анти-паттерном для широкого класса систем / условий и др.
- **Как бороться:**
  - Использовать таймеры и прерывания
  - Использовать планировщик
    - Блокирующие вызовы: wait, select, etc.
    - Мьютексы, условные переменные…

Error hiding

- **Прятать сообщение об ошибке за “дружественным сообщением”, либо подавлять вообще**
  - Давайте родитель также будет вызывать виртуальные методы, переопределенные в потомке, которые будут вызывать методы предка, и т.д.
- **Как бороться:**
  - Пусть программа падает…
  - Логгирование всех ошибок

```
int main() {
        try {
        ...
        } catch (...) {
                std::cout << "Oops\n";
        }
        return 0;
}
```

Magic numbers, magic strings…

- **Использование числовых / литеральных констант в коде**
  - Усложняет модификацию системы, потенциально повышает хрупкость
  - Ухудшает читабельность кода, реализующего сложную логику
- **Как бороться:**
  - Использовать дефайны, константы, конфиги и т.д.

Bad var/func/class names

- **Использование запутанных, “не говорящих” имён в коде**
  - Ухудшает читабельность кода, реализующего сложную логику

God object

- **В системе появляется один объект, который управляет всеми вычислениями, остальные -- передают ему данные. Публичный интерфейс -- “сделать всё(...)”**
  - Привычка структурного программирования -- разделение данных и кода
  - Часто обусловлено разрастанием объекта от proof-of-concept в сжатые сроки
  - Детектируется по размеру классов / количеству методов, метрикам сопряжения
  - Исключения -- обёртки над легаси-компонентами
- **Как бороться:**
  - Разделение функциональности
  - Не писать код до проектирования архитектуры

Swiss army knife

- **В системе появляется объект с чрезвычайно сложным интерфейсом, который делает всё, но при этом не берёт всё управление на себя, в отличие от God object**
  - Часто появляется в библиотеках вследствие попыток производителя сделать свою технологию более широко применимой
  - Инкапсуляция сложности приносится в жертву гибкости →бессмысленная абстракция
- **Как бороться:**
  - Разделение функциональности
  - Не писать код до проектирования архитектуры

God object / Swiss army knife refactoring

- **Избавление от God object’**а -- целая серия мероприятий по рефакторингу
- **Разделить методы класса на группы, соответствующие контрактам (функциям и условиям), выполняемым God object.**
- **Поискать среди уже написанных классов более подходящие для каждой группы методов**
- **При необходимости -- создать новые классы, в соответствии с принципом единственной ответственности “S” SOLID**
- **Убрать непрямые зависимости**
  - Если объекты A, B включены в C, то, возможно, A включен в B, а B в C.

- **В случае со Swiss army knife:**
  - Примерно то же самое, но:
    - Перепроектировать интерфейс согласно принципу “I” сегрегации интерфейсов SOLID [даже если визуально всё ОК]

Lava flow

- **Мертвый или недостижимый код (или вполне себе достижимый :)), “незакрытый технический долг” застыл в системе, как поток лавы**
  - Проявляется зачастую как “костыль”, который “работает -- не трогай”.
  - Разработчик, который писал это -- уволился 5 лет назад, а оставшиеся не имеют идей, как это исправлять (на то есть, как правило, объективные причины)
- **Закомментированный код, TODO, большие методы, невнятные интерфейсы**
  - Зачастую такие вещи запрещены стандартом кодирования, принятым командой, но …

…

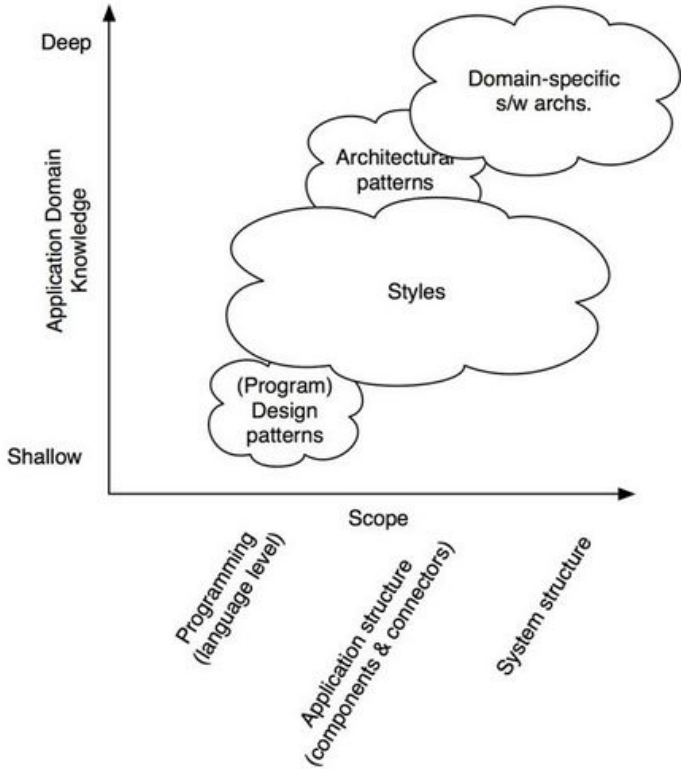
Архитектурные шаблоны и стили

Архитектурный стиль -- набор решений, которые:

- Применимы в выбранном контексте построения программного решения
- Задают скорее ограничения на принимаемые архитектурные решения, специфичные для определённых систем
- Приводят к желаемым свойствам разрабатываемой системы

Архитектурный шаблон -- именованный набор ключевых решений по организации подсистем, применимых для повторяемых технических задач проектирования в различных контекстах и предметных областях. Это более “тактические” примеры архитектуры, но всё же более глобальные, чем просто паттерны проектирования.

Разделение, конечно же, довольно условное.



N. Medvidovic

Taylor, R. N., Medvidovic, N.,, Dashofy, E. M. (2007). Software Architecture: Foundations, Theory and Practice. Addison-Wesley.

Архитектурные шаблоны и стили

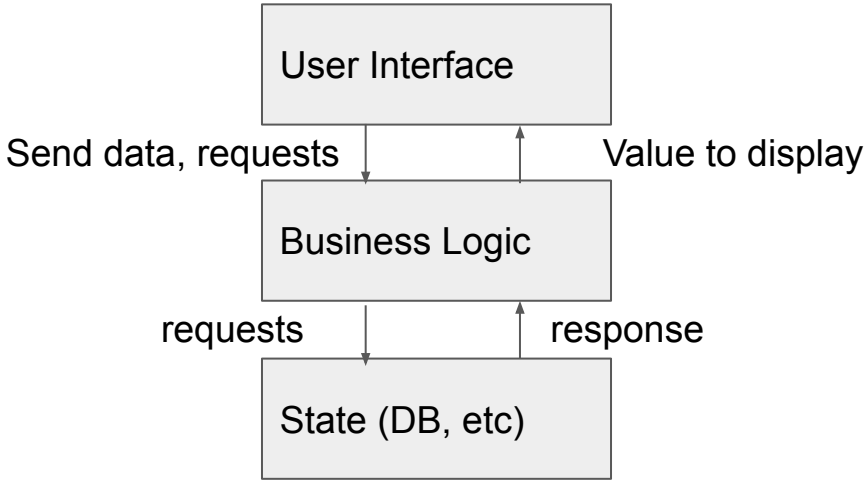
Архитектурный стиль -- набор решений, которые:

- Применимы в выбранном контексте построения программного решения
- Задают скорее ограничения на принимаемые архитектурные решения, специфичные для определённых систем
- Приводят к желаемым свойствам разрабатываемой системы

Архитектурный шаблон -- именованный набор ключевых решений по организации подсистем, применимых для повторяемых технических задач проектирования в различных контекстах и предметных областях. Это более “тактические” примеры архитектуры, но всё же более глобальные, чем просто паттерны проектирования.

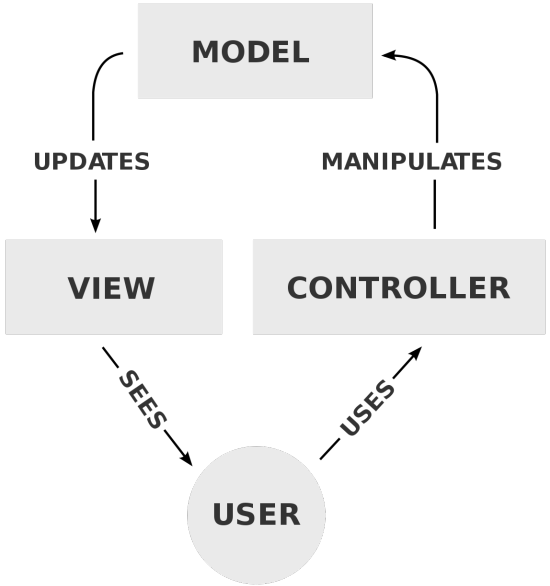
Разделение, конечно же, довольно условное.

State-Logic-Display (трёхзвенная архитектура)



- Один из простейших примеров “слоёной” архитектуры, в которой:
  - Интерфейс пользователя отделён от бизнес-логики
  - Бизнес-логика взаимодействует с хранилищем и не хранит состояние
  - Состояние хранит только хранилище
  - Масштабируемость следует “по построению”, потому как у бизнес-логика stateless
- Используется в: бизнес-приложениях, многопользовательских играх, веб

Model-View-Controller



- Разделить систему согласно ф-ти на данные, представление и взаимодействие
- Минимум 3 компонента, каждый из которых будет:
  - Сопряжён с другими через стандартизированный “однонаправленный” интерфейс
  - Способен разрабатываться в независимости от других (“mock’и”)
  - Не обязан быть единственным
  - Реализуется по правилам и техникам, принятым именно для его реализации

Model-View (Qt)

View сам модифицирует модель, получает от неё сообщения, обновляется

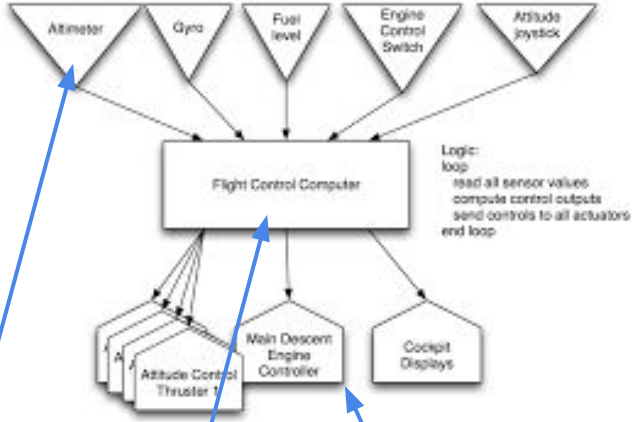
+ Проще в реализации
- Неструктурированное взаимодействие, контроллера нет (как единственного места, где исполнять все команды и т.д.)

Model-View-View Model (Dotnet WPF, Android Studio)

- Модель это модель
- View это View
- View-model -- промежуточный компонент, представляющий данные модели для View

Model-View-Presenter

Sense-Compute-Control



- Считать данные сенсоров
- Рассчитать управляющее воздействие
- Передать управляющее воздействие на актуаторы

Применяется в робототехнике

Факт, демонстрируемый данным шаблоном: архитектурный шаблон -- это только вершина айсберга…

И в некотором смысле это -- детали реализации, а нас больше может интересовать алгоритмическая начинка.

https://www.ics.uci.edu/~taylor/classes/211/DesignAndArchitecture.pdf

Архитектурные стили

- Именованные концепции архитектурных решений
- Менее узкоспециализированные, чем архитектурные шаблоны

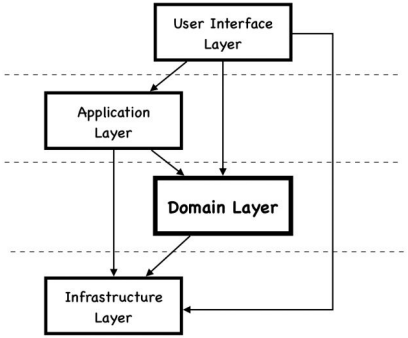
- Переиспользование архитектуры
  - Хорошо известные и изученные решения для разных задач
- Переиспользование кода
- Упрощение взаимодействия разработчиков
- Упрощение интеграции приложений
- Специфичные для стиля методы анализа
  - Возможны благодаря ограничениям на структуру системы
- Специфичные для стиля методы визуализации

Монолитный стиль

- Каждый с каждым
- Теоретически, мы достигаем

Слоистый стиль

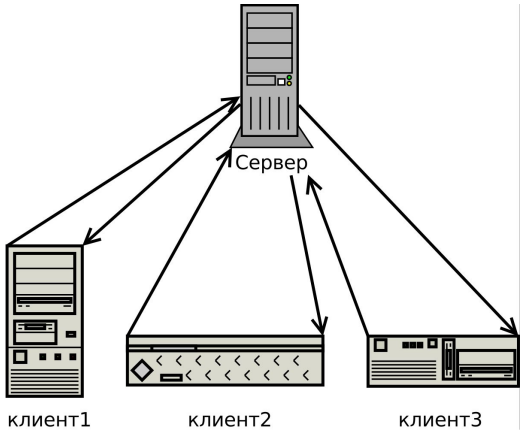
- Система делится на **слои**
  - **Слой взаимодействует с** соседними, либо в обе стороны, либо только в 1 сторону



\+ Повышение уровня абстракции
\+ Легкость расширения
\+ Изменения на каждом уровне затрагивают максимум 2 соседних
\+ Возможность последовательной разработки и отладки
\+ Возможны различные реализации уровня, совместимые с интерфейсами
\- Не всегда применим
\- Проблемы с производительностью

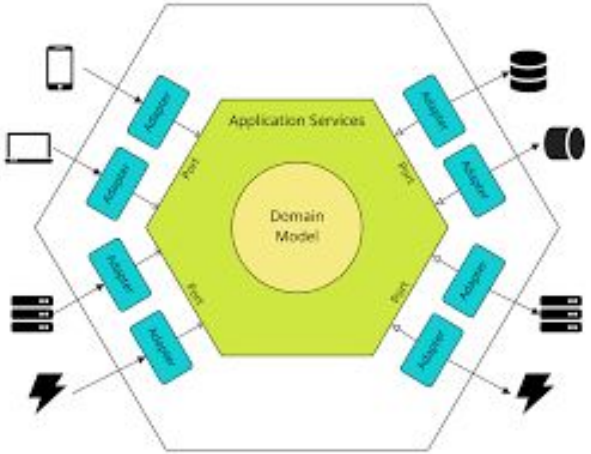
Клиент-сервер

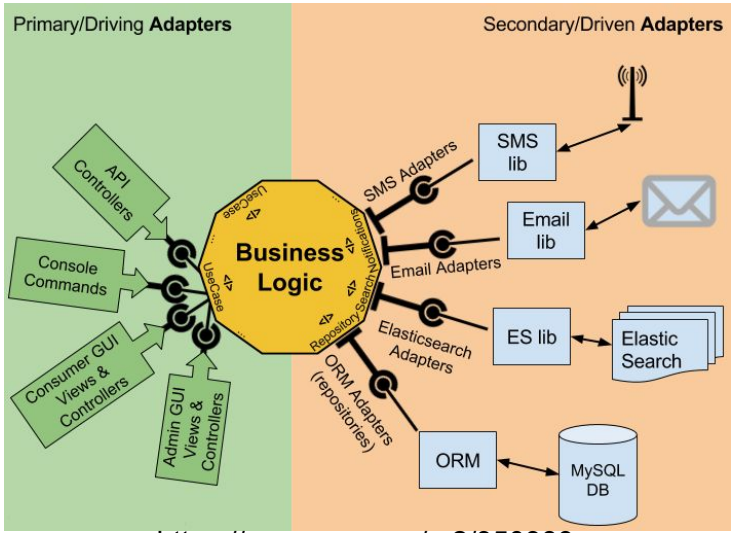
Компоненты -- клиенты и серверы
Серверы не знают о ещё не подключенных клиентах (как правило)
Клиенты не знают друг о друге
Соединители -- сетевые протоколы или просто IPC



Гексагональная архитектура (“порты и адаптеры”)

+ Изоляция механизмов доставки
+ Изоляция вспомогательных механизмов
  - Библиотеки, и др.
+ Легкость тестирования, mocks
+ “Чистая” бизнес логика и модель предметной области
  + Максимальная простота, возможность валидации и преобразования данных
- Тяжеловесна (от необходимости отделять средства доставки от бизнес-логики)
- Неподробна
- Что делать с фреймворками?



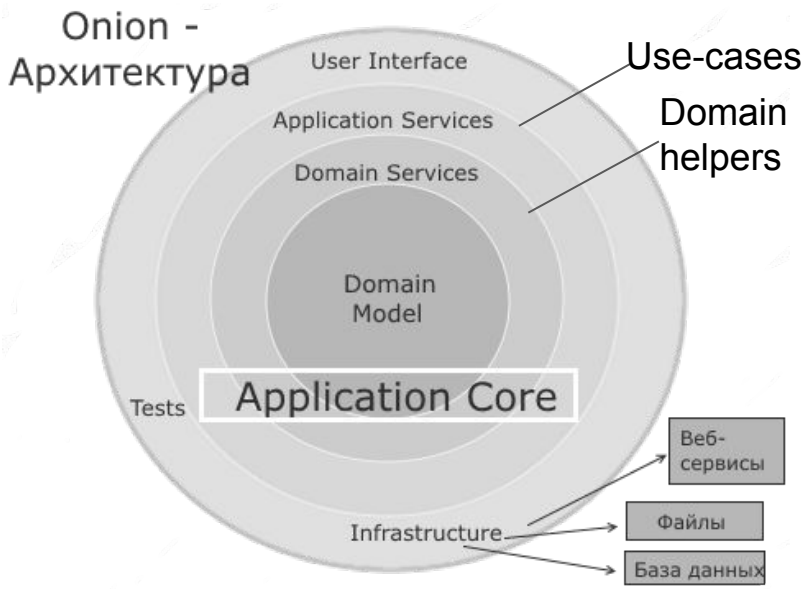


https://www.pvsm.ru/c-2/350389

Луковая архитектура (уточнение гексагональной)

Определяет (уточняет) внутреннюю структуру ядра
Внутренние слои не знают о внешних, модель предметной области не знает ни о ком
Внутренние слои определяют интерфейсы, внешние -- их реализуют
Уровневость нестрогая -- слой может использовать все слои под ним
Репозитории, БД -- внешняя зависимость (в самом начале было не так)

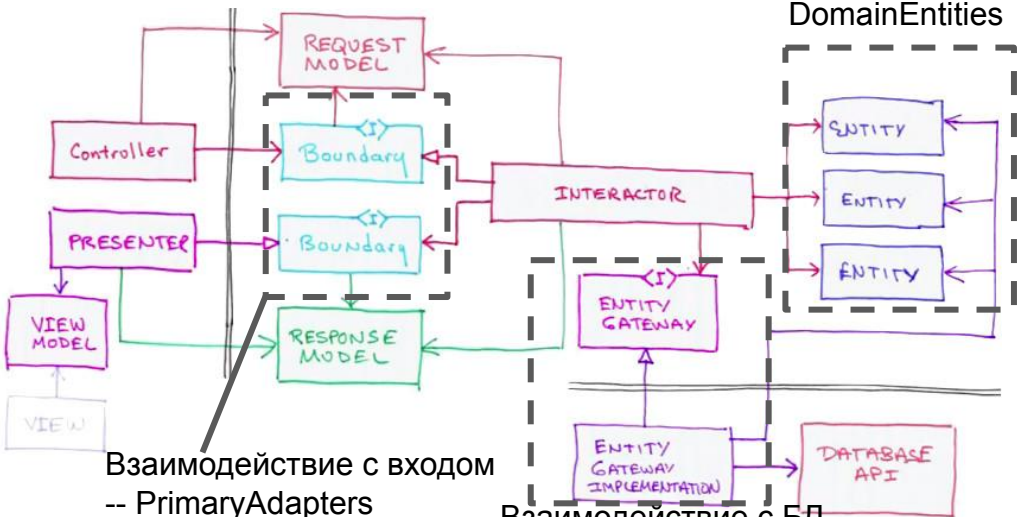
\+ Уточнение, большая ясность
\- Почти те же, что и у гексагональной



“Чистая” архитектура (уточнение луковой)

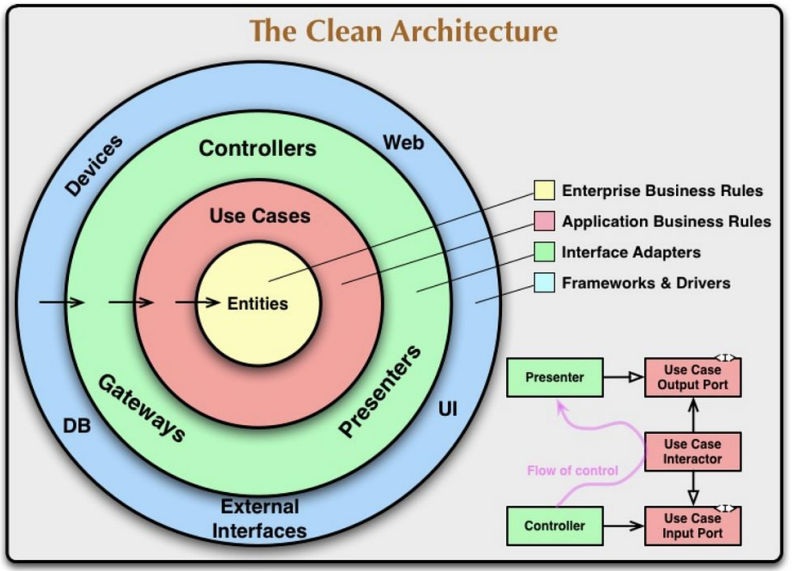
Определяет (уточняет) потоки управления
Добавляет описание набора граничных интерфейсов, которые описывают взаимодействие с внешним “не чистым” миром.

\+ Уточнение, большая ясность
\- Схема очень сложна



Взаимодействие с входом
-- PrimaryAdapters

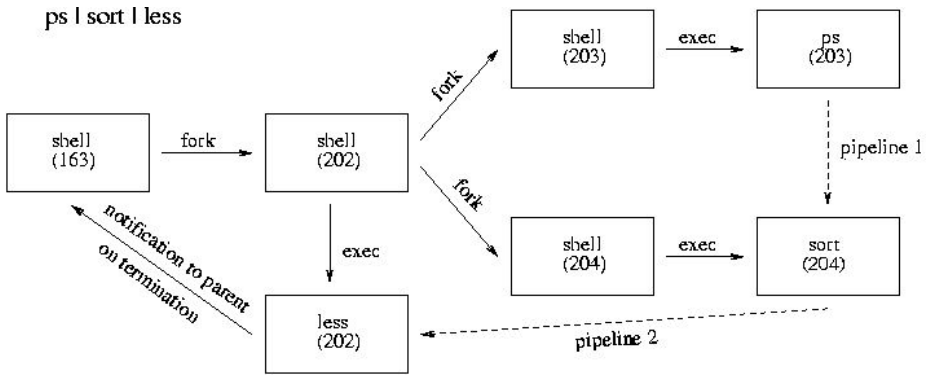
Взаимодействие с БД
-- SecondaryAdapters



https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
https://softwareengineering.stackexchange.com/questions/388578/clean-architecture-is-the-input-boundary-necessary

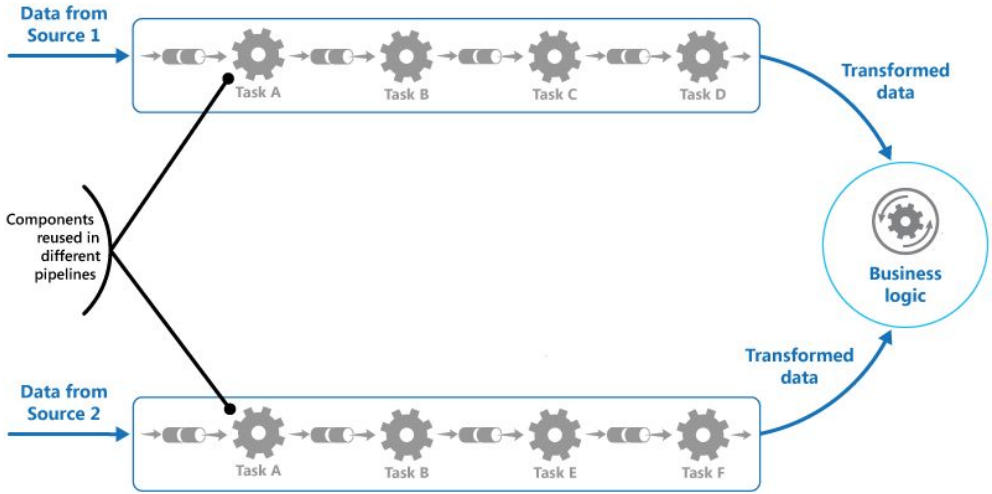
Пакетная обработка, конвейер

- Вспомним конвейеры с прошлого семестра…



Каналы и фильтры

Улучшение концепции пакетной обработки
Фильтры (как правило) не пересоздаются, а существуют
Фильтры (как правило) типизируемые, а каналы -- ограничены по пропускной способности

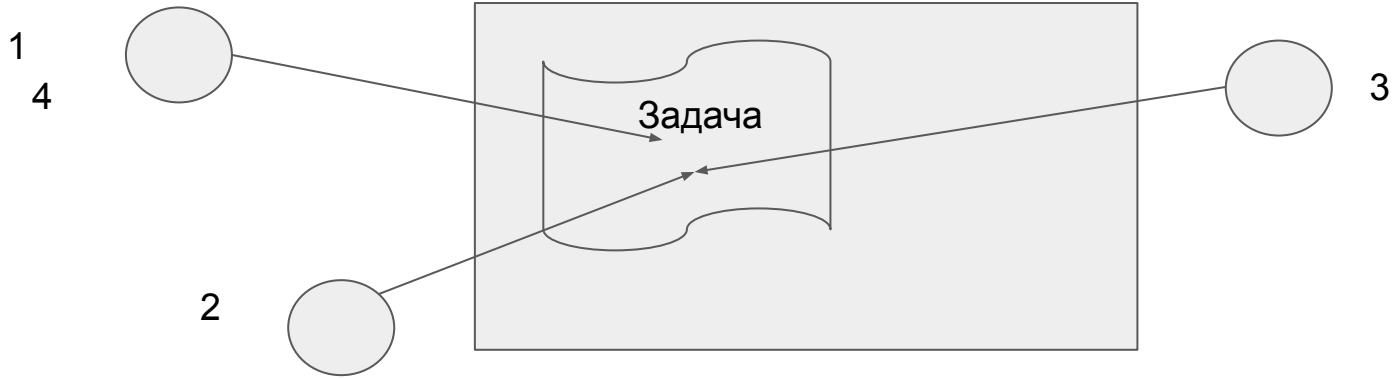


\+ Простота
\- Возможны сложности с откликом для пользователя
\- Узкие места определяют производительность всей системы

Blackboard

Агенты без состояния обрабатывают центральную структуру данных, хранящую состояние, либо по порядку, либо пока есть что обрабатывать.
Используется в:

- Компиляторных оптимизаторах
- Графовых грамматиках
- Всяких решателях…



Событийный стиль

- Оповещение о событии вместо явного вызова метода
  - Слушатети: подписываются
  - Система при наступлении события сама вызывает все зарегистрированные методы слушателей
- Компоненты имею 2 вида интерфейсов -- методы и события
- Соединения:
  - Явный вызов метода
  - Неявный вызов по наступлению события
- Инварианты:
  - Источники событий не знают, кто на них реагирует
  - Нет предположений о том, как обрабатывается событие (и будет ли обработано вообще)

\+ Переиспользуемость компонент через подписки
  \+ Очень низкая связность между компонентами
\- Неинтуитивная структура программы (для “сырого стиля”)
\- Компоненты не управляют последовательностью вычислений
\- Непонятно, кто реагирует на запрос
\- Тяжело отлаживать

Событийный стиль: Push-subscribe

- Уточнение событийного стиля
  - Компоненты разделены на подписчиков и издателей

+ Наводит порядок в событийном стиле
  + Очень низкая связность между компонентами

Событийная шина

Компоненты общаются между собой только по шине, по которой летят события. Это -- централизованное место (+ балансировщик нагрузки и маршрутизатор).

Компоненты могут не иметь своего состояния вообще, а строить его по сообщениями с шины.

Шина используется как единый источник истины для очень распределенной системы

peer-to-peer

- Система из одинаковых компонентов, которые соединены сетевыми протоколами