

## Вопросы

1. Понятие архитектуры, профессия “Архитектор”.
2. Архитектурные виды.
3. Роль архитектуры в жизненном цикле ПО.
4. Понятие декомпозиции. Модульность, связность, сопряжение, сложность.
5. Понятия класса и объекта, абстракция, инкапсуляция, наследование.
6. Принципы выделения абстракций предметной области.
7. Принципы SOLID.
8. Закон Деметры. Некоторые принципы хорошего объектно-ориентированного кода.
9. Моделирование, визуальные модели, виды моделей.
10. Язык UML. Проектирование структуры системы, диаграммы классов.
11. Диаграммы объектов, диаграммы пакетов UML.
12. Диаграммы компонентов, диаграммы развёртывания UML.
13. Диаграмма случаев использования UML.
14. Диаграмма активностей UML.
15. Диаграммы конечных автоматов UML.
16. Диаграммы последовательностей UML.
17. Диаграммы коммуникации UML.
18. Диаграммы составных структур, коопераций, временные диаграммы.
19. Диаграммы обзора взаимодействия, диаграммы потоков данных.
20. Диаграммы “Сущность-связь”.
21. Понятие архитектурного стиля, трёхзвенная архитектура,
22. Model-View-Controller, Sense-Compute-Control.
23. Структурный и объектно-ориентированный стили, слоистые архитектурные стили.
24. Пакетная обработка, каналы и фильтры, Blackboard.
25. Стили с неявным вызовом, Publish-Subscribe.
26. Peer-to-peer, C2, CORBA.
27. Понятие Domain-Driven Design, единый язык, изоляция предметной области.
28. Основные структурные элементы модели предметной области.
29. Паттерны “Агрегат”, “Фабрика”, “Репозиторий”.
30. Моделирование ограничений, паттерн “Спецификация”.
31. Паттерн “Компоновщик”.
32. Паттерн “Декоратор”.
33. Паттерн “Стратегия”.
34. Паттерн “Адаптер”.
35. Паттерн “Прокси”.
36. Паттерн “Фасад”.
37. Паттерн “Мост”.
38. Паттерн “Приспособленец”.
39. Паттерн “Спецификация”.
40. Паттерн “Фабричный метод”.
41. Паттерн “Шаблонный метод”.
42. Паттерн “Абстрактная фабрика”.
43. Паттерн “Одиночка”.

44. Паттерн "Прототип".
45. Паттерн "Строитель".
46. Паттерн "Посредник".
47. Паттерн "Команда".
48. Паттерн "Цепочка ответственности".
49. Паттерн "Наблюдатель".
50. Паттерн "Состояние".
51. Паттерн "Посетитель".
52. Паттерн "Хранитель".
53. Архитектура распределённых систем: понятие распределённой системы, типичные архитектурные стили.
54. Межпроцессное сетевое взаимодействие, модель OSI, стек протоколов TCP/IP, сокет, протоколы "запрос-ответ".
55. Удалённые вызовы процедур (RPC), удалённые вызовы методов (RMI). Protobuf, gRPC.
56. Веб-сервисы, SOAP. WCF.
57. Очереди сообщений, RabbitMQ.
58. REST.
59. Микросервисы, peer-to-peer.
60. Развёртывание и балансировка нагрузки, Docker.

# 1. Понятие архитектуры, профессия “Архитектор”.

**Архитектура** - совокупность важнейших решений об организации программной системы

- Эволюционирующий свод знаний
- Разные точки зрения
- Разный уровень детализации

Нужна, чтобы проект не развалился из-за собственной сложности. Совершенно не интересна заказчику.

**Архитектор** - специально выделенный человек (или группа людей), отвечающий за:

- разработку и описание архитектуры системы
- доведение её до всех заинтересованных лиц
- контроль реализации архитектуры
- поддержание её актуального состояния по ходу разработки и сопровождения

**Профессиональный стандарт** - создание и сопровождение архитектуры программных средств.

**Трудовые функции** - создание вариантов архитектуры, их документирование, оценка требований, различный контроль. А именно:

- Создание вариантов архитектуры программного средства
- Документирование архитектуры
- Оценка требований
- Оценка и выбор варианта архитектуры
- Оценка возможности создания архитектурного проекта
- Контроль реализации и сопровождения
- Утверждение и контроль методов и способов взаимодействия программного средства со своим окружением
- Реализация программных средств (в основном контроль и анализ)
- Модернизация программного средства и его окружения

У архитектора более широкий кругозор, но меньше глубина знаний, чем у разработчика. но часто их роли взаимозаменяемы. Архитектор не должен находиться в башне из слоновой кости, то есть он должен участвовать в проекте. А еще архитектор иногда общается с руководством и заказчиком, поэтому ему нужны социальные навыки. Архитектор должен оставлять реализационные аспекты программистам.

Флекс от Олега:

<http://files.catwell.info/misc/mirror/2003-martin-fowler-who-needs-an-architect.pdf>

## 2. Архитектурные виды.

**Неправильные решения на этапе проектирования очень дороги.**

Многое зависит от интуиции и вкуса архитектора, выделяют различные архитектурные виды (views), точки зрения, стили.

Архитектура ПО обычно содержит несколько видов, которые аналогичны различным типам чертежей в строительстве зданий. Архитектурный вид состоит из 2 компонентов - элементов и отношений между элементами.

Архитектурные виды можно поделить на 3 **основных типа**:

1. Модульные виды (англ. *module views*) — показывают систему как структуру из различных программных блоков.
2. Компоненты-и-коннекторы (англ. *component-and-connector views*) — показывают систему как структуру из параллельно запущенных элементов (компонентов) и способов их взаимодействия (коннекторов).
3. Размещение (англ. *allocation views*) — показывает размещение элементов системы во внешних средах.

Виды из IEEE 1016:

1. Контекст — фиксирует окружение системы
2. Композиция — описывает крупные части системы и их предназначение
3. Логическая структура — структура системы в терминах классов, интерфейсов и отношений между ними
4. Зависимости — определяет связи по данным между элементами
5. Информационная структура — определяет персистентные данные в системе
6. Использование шаблонов — документирование использования локальных паттернов проектирования
7. Интерфейсы — специфицирует информацию о внешних и внутренних интерфейсах
8. Структура системы — рекурсивное описание внутренней структуры компонентов системы
9. Взаимодействия — описывает взаимодействие между сущностями
10. Динамика состояний — описание состояний и правил переходов между состояниями
11. Алгоритмы — описывает в деталях поведение каждой сущности
12. Ресурсы — описывает использование внешних ресурсов

Для всего этого есть различные диаграммы UML. Ничто из этого не обязательно. Выглядит все это [так](#).

### 3. Роль архитектуры в жизненном цикле ПО.

Похоже, архитектура влияет на все этапы жизненного цикла. Есть компании с одной командой архитекторов, которые занимаются проектами по очереди, но сейчас склоняются к мысли, что создание архитектуры - непрерывный процесс.

Архитектор принимает участие в сборе требований. **Требования** к системе бывают функциональные (что делать) и нефункциональные (как делать), например на:

- Эффективность
- Масштабируемость
- Удобство использования
- Надежность (наработка на отказ, время восстановления, поведение при сбое)
- Безопасность
- Сопровождаемость

Нефункциональные требования сложнее.

А еще есть ограничения - *технические* (на iPhone не напишешь на хаскеле) и *бизнес-ограничения* (сроки горят, через месяц будет поздно).

**Архитектурный дрейф** - имплементация отличается от изначального дизайна (*prescriptive architecture*). Влияет только на часть системы. Не вносят сильного хаоса.

**Архитектурная эрозия** - новые решения в архитектуре, отличающиеся от изначального дизайна. Нарушение именно основных принципов. Более серьезное нарушение. Вносят сильный хаос.

Любые исправления в системе увеличивают ее энтропию. Этот процесс можно контролировать, например, с помощью регулярных рефакторингов.

Пример - Hadoop, который as-built сильно отличается от as-designed.

## 4. Понятие декомпозиции. Модульность, связность, сопряжение, сложность.

**Существенная сложность** (*essential complexity*) - присуща самой проблеме. Можно контролировать, нельзя избавиться.

**Случайная сложность** (*accidental complexity*) - сложность, привнесенная способом решения. От нее можно и нужно избавляться.

**Декомпозиция** — разбиение иерархичной системы на компоненты, каждый из которых состоит из более мелких компонентов и т.д., при этом про каждый компонент можно более-менее осмысленно рассуждать в отдельности;

**Абстрагирование** — выделение общих свойств компонентов, их классификация, рассуждение не о конкретных элементах системы, а об их типах.

Сложные системы появляются из простых, поэтому надо даже простую систему задумывать, как большую в будущем.

Проектирование бывает **восходящее** (сначала разрабатываем компоненты, потом из них более сложные компоненты) и **нисходящее** (берем общую систему с заглушками, потом решаем подзадачи пока не придем к окончательному решению).

Восходящее проектирование может использоваться для исследовательских работ, если точно не понятно, что хочется получить в конце. Также можно использовать восходящее проектирование, если есть несколько задач, использующих одинаковые модули.

**Модули** - структурные единицы кода, которые соответствуют подзадачам, на которые разбита система. Пример - пара из .c и .h файлов. У них есть интерфейс и реализация. Основные принципы - четкая декомпозиция, минимизация интерфейса, single responsibility, отсутствие не очевидного влияния на другие модули, независимость, сокрытие данных.

Надо соблюдать баланс в размере модуля - слишком маленькие модули могут усложнять интеграцию и нивелировать пользу от разбиения на подзадачи. А еще есть правило "7 +/- 2" - столько сущностей в голове может удержать человек.

**Сопряжение** (*Coupling*) — мера того, насколько взаимосвязаны разные модули в программе.

**Связность** (*Cohesion*) — мера того, насколько взаимосвязаны функции внутри модуля и насколько похожие задачи они решают.

Целью при проектировании является слабое сопряжение и сильная связность.

## 5. Понятия класса и объекта, абстракция, инкапсуляция, наследование.

**Объект** — это маленький компьютер с состоянием и допустимыми операциями (с)

Thinking in Java

А также область памяти, содержащая значение, а еще куча других определений.

**Свойства** — состояние, поведение и идентичность (возможность однозначной идентификации).

**Инвариант** — набор логических условий, которые должны исполняться все время жизни объекта. Объект обязан не давать возможности нарушить свой инвариант. Бывают неявные инварианты - при увеличении одной из сторон прямоугольника вдвое площадь увеличится вдвое. Квадрат его не выполняет.

**Класс** — тип объекта. Определяет структуру данных и методы. Класс - сущность compile time, объект - сущность runtime.

**Абстракция** — выделение существенных признаков объекта, отличающих его от остальных объектов, с точки зрения наблюдателя. У одного объекта может быть несколько абстракций. Пример абстракции - алгебраические структуры.

**Инкапсуляция** — разделение интерфейса абстракции и ее реализации. Также защищает инварианты от порчи.

**Наследование** — объект типа-потомка является одновременно объектом типа-предка, поэтому может использоваться везде, где может использоваться предок. Потомок наследует все инварианты предка и не вправе их нарушать. Вместо наследования для переиспользования кода нынче модно использовать композицию - выносим общий код в отдельный класс и имеем его как поле.

## 6. Принципы выделения абстракций предметной области.

- Определение объектов и их атрибутов
- Определение действий, которые могут быть выполнены над каждым объектом (назначение ответственности)
- Определение связей между объектами
- Определение интерфейса каждого объекта

Результат — набросок системы. Например, диаграмма классов.

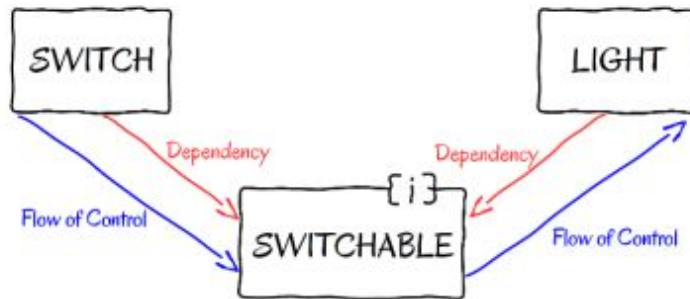
Источники абстракций (помимо предметной области):

- Изоляция сложности — прячем сложность от остальной системы
- Изоляция возможных изменений
  - бизнес-правила (e.g. алгоритмы предметной области)
  - Зависимость от оборудования и ОС
  - ввод-вывод
  - смена компилятора (нестандартные возможности языка)
  - архитектура
  - 3rd party
- Изоляция служебной функциональности (фабрики, сервисы и т.д.)



## 7. Принципы SOLID.

1. **Single responsibility principle** — каждый класс должен делать что-то одно. А еще важно к нему писать комментарии о том, что он делает. А также вся соответствующая функциональность должна быть в этом классе (плохо иметь “класс для работы со строками” и “другой класс для работы со строками”)
2. **Open/closed principle** — абстракция (класс, модуль, функция) должна быть открыта для расширения, но закрыта для изменения. Для добавления новой функциональности нужно использовать наследование или заранее подготовленные точки расширения.
3. **Liskov Substitution Principle (принцип подстановки Барбары Лисков)** — Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом. Классы-потомки должны реализовывать интерфейсы классов предков (как чисто синтаксически, так и семантически — делать то, что ожидается от предка) и, про что часто забывают, выполнять все инварианты классов-предков, явные или неявные. Хороший пример проблем с принципом подстановки — проверяемые исключения в Java. Методы класса-предка могли задекларировать исключения, которые они могут бросать, тогда потомок, переопределяющий эти методы, имел право бросать только эти исключения. Почему — вызывающий мог использовать try/catch для обработки ошибок, и если там были написаны catch-блоки для исключений предка, а потомок бросал что-то новое, это ломало вызывающего и тем самым нарушало принцип подстановки. Но все ситуации проверить невозможно - поэтому больше не существует проверяемых исключений.
4. **Interface Segregation Principle** — клиенты не должны зависеть от методов, которые они не используют. Есть смысл делать разные интерфейсы на разные группы методов (по смыслу, а не механически). Например на модем с методами connect() и send() надо два интерфейса. Потому что сторонний модуль, который делает только send() должен будет меняться, если мы поменяем параметры connect().
5. **Dependency Inversion Principle** — модули верхних уровней не должны зависеть от модулей нижних уровней, оба типа модулей должны зависеть от абстракций.



Здесь Switch мог бы содержать Light и делать `light.on()/off()`. Но тогда он

1. Может управлять только лампочкой
2. Меняем лампочку = перекомпилируем
3. Надо знать, что за лампочка, чтобы понять, что делает switch

Решение - интерфейс. Теперь switch не знает про лампочку. Получает лампочку либо параметром в конструктор, либо через метод-сеттер (Dependency Injection). Этим иногда занимаются целые Inversion of Control библиотеки.

## 8. Закон Деметры. Некоторые принципы хорошего объектно-ориентированного кода.

Для общей эрудиции - the Demeter project was named after Demeter because we were working on a hardware description language Zeus and we were looking for a tool to simplify the implementation of Zeus. We were looking for a tool name related to Zeus and we chose a sister of Zeus: Demeter.

Закон Деметры не является частью SOLID, но это хорошее правило. Кратко звучит как “не разговаривай с незнакомцами”. А именно - объект А не должен иметь возможность получить непосредственный доступ к объекту С, если у объекта А есть доступ к объекту В, и у объекта В есть доступ к объекту С.

```
book.pages.last.text // Плохо
book.pages().last().text() // Лучше, но тоже не супер
book.lastPageText() // Идеально
```

Вызовы по цепочке раскрывают внутреннюю структуру данных класса. Например, тут нам обязателен метод last() (а другие не подойдут). Если мы поменяем представление данных посередине, придется менять все места использования. Но надо знать меру - код вида book.firstPageText(), book.secondPageText(), book.thirdPageText() и т.д. гораздо хуже. В таком случае можно все же представить коллекцию pages().

И не надо путать с законом Деметры вызовы по stream API, например. Это называется Fluent API. Похожая идея используется в паттерне “Строитель”. Тут все ок, потому что цепочка возвращает один и тот же объект, не раскрывая его структуры.

Краткий обзор “Чистого кода” Р. Мартина:

### 1. Абстракция

#### ○ Шрифт

- currentFont.size = 16 — прямое присвоение полю, ужасно. Во-первых, шрифт не может поддерживать свой инвариант и изменить внутреннее представление размера. Во-вторых, тут написано “размеру присвоить 16”. Окей, чего 16?
- currentFont.size = PointsToPixels(12) — чуть лучше. Теперь размер, видимо, измеряется в пикселях.
- currentFont.sizeInPixels = PointsToPixels(12) — ещё чуть лучше. Теперь про размер явно написано, что он измеряется в пикселях.
- currentFont.setSizeInPixels(sizeInPixels) — совсем хорошо. Нам-то какое дело, в чём внутри у шрифта измеряется размер, мы хотим его выставить в тех единицах измерения, которые нам удобны.

```
public class Program {
    public void initializeCommandStack() { ... }
    public void pushCommand(Command command) { ... }
    public Command popCommand() { ... }
    public void shutdownCommandStack() { ... }
    public void initializeReportFormatting() { ... }

```

```

    public void formatReport(Report report) { ... }
    public void printReport(Report report) { ... }
    public void initializeGlobalData() { ... }
    public void shutdownGlobalData() { ... }
}

```

○

- Надо поделить на 3 класса - — стек команд, управлялка отчётами и хранилище глобальных данных. И добавить четвёртый класс, который бы управлял этими тремя классами.

```
public class Point {
    public double x;
    public double y;
}

```

против

```
public interface Point {
    double getX();
    double getY();
    void setCartesian(double x, double y);
    double getR();
    double getTheta();
    void setPolar(double r, double theta);
}

```

○

- Второй лучше (скрывает внутреннюю реализацию), но платим за это строками кода
- Итого - задача абстракции упростить понимание, а не уметь делать много всего. Например, список сотрудников не обязательно должен наследоваться от списка.
- Общие рекомендации

- Учитывайте противоположные методы (add/remove), в дальнейшем могут понадобиться
- Не возвращайте null
- По возможности делайте некорректные состояния невыразимыми в системе типов
- Все, что можно проверить компилятором, должно быть написано так, чтобы проверялось компилятором

## 2. Инкапсуляция

- public полей не бывает - надо делать геттеры и сеттеры. Даже если это класс для передачи данных.
- Класс не должен знать о клиентах
- Читаемость важнее удобства написания
- Не надо делать семантические нарушения инкапсуляции (даже если метод делает коннект, если его еще нет, неправильно на него надеяться, нужно сначала сделать коннект самим)
- Protected и package private полей тоже нет - для них можно сделать отдельный интерфейс

## 3. Наследование

- Говорят, что наследование не нужно. Вместо него предлагается использовать агрегацию.

## 4. Инициализация

- Инициализируйте в конструкторе все, что можно
- Синглтоны надо использовать осторожно, так как это по сути глобальные переменные
- <всякие прочие штуки>

## 5. Мутабельность

- Ее надо избегать
- <далее советы про это>

## 6. Скорость не так важна, как читаемость

## 7. Общие рекомендации

- Fail fast - обрабатывать малейшие ошибки, в том числе валиться
- Подробная документация
- Статические проверки > проверки в рантайме
- Юнит-тесты
- CI
- Не бойтесь все переписать - первые версии часто выкидываются и способствуют пониманию системы

## 9. Моделирование, визуальные модели, виды моделей.

Модель — упрощенное подобие объекта / явления, созданное для изучения его свойств.

Свойства моделей:

1. Содержат меньше информации, чем реальность
2. Субъективны (выделяем существенные свойства)
3. Помогают управлять сложностью
4. Позволяют проанализировать и протестировать систему до реализации
5. Имеют цель
6. Общение

Стоимость создания и поддержания модели не должна превышать её пользы.

Виды моделей:

1. Обычный текст
  - Неформален, неоднозначен, не строг
  - Многословен
  - Сложно обрабатывать автоматически
2. Неформальные графические модели: диаграммы powerpoint, рисунки
  - + Гибкая нотация, красивые
  - Неформальны, неоднозначны, не строги
  - Сложно обрабатывать автоматически
3. UML и SysML — несколько нотаций.
  - + Общеприняты, широкая поддержка
  - Нет строгой семантики
  - Сложно расширять
4. Текстовые формальные языки (AADL)
  - + Продвинутые инструменты анализа
  - + Хороши для моделирования систем реального времени
  - Многословны, детальны, сложны

Важные моменты визуальных моделей:

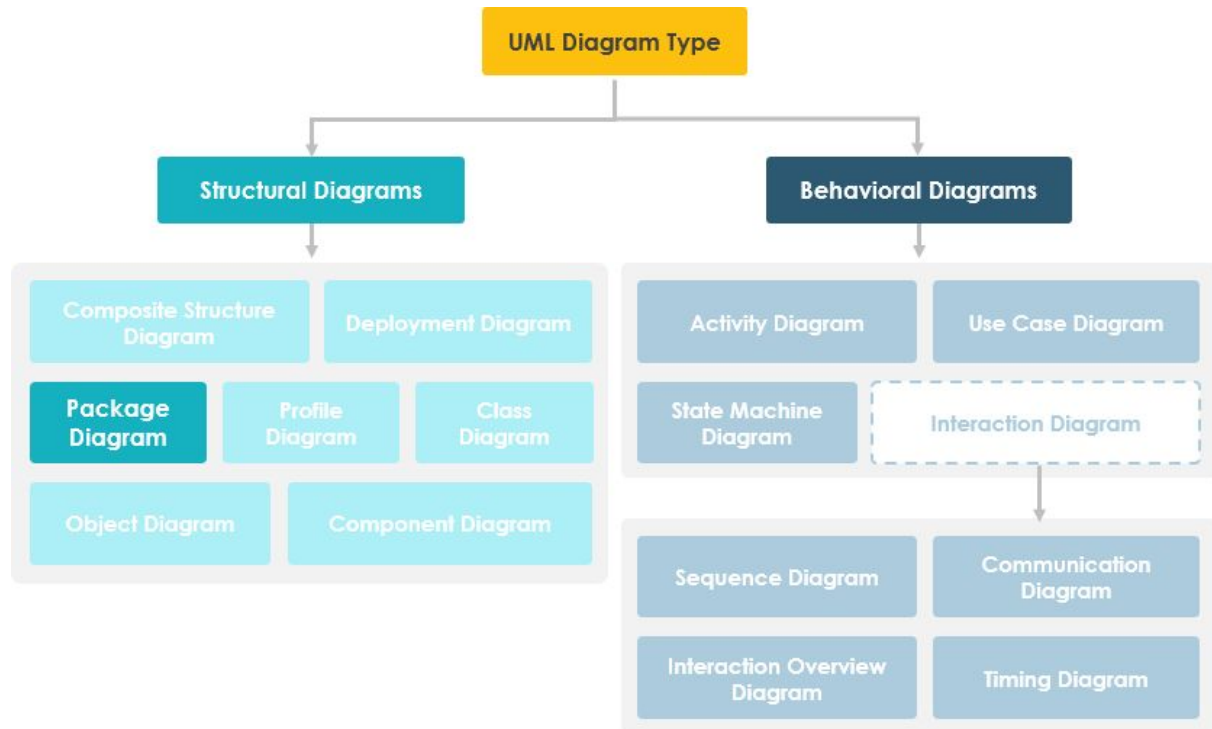
- **Метафора визуализации** — договоренность о представлении сущностей языка.
- **Точка зрения моделирования** — что (какой аспект) и для кого моделируется.

**Семантический разрыв** — невозможно полностью покрыть моделью исходную систему.

## 10. Язык UML. Проектирование структуры системы, диаграммы классов.

UML (Unified Modeling Language) — семейство графических нотаций. Этот открытый стандарт позволяет достигнуть соглашения в обозначениях, используемых при визуальном моделировании.

UML диаграммы часто используются для проектирования структуры системы: поиска решения, удовлетворяющего функциональным требованиям, целям и ограничениям.



Все диаграммы делятся на

1. Структурные — структура системы времени компиляции
2. Поведенческие — поведение системы во время работы

Четкого разделения на типы диаграмм по сути нет: есть общая нотация UML. Это можно использовать для совмещения различных диаграмм. Однако не стоит мешать структурные и поведенческие диаграммы.

**Диаграмма классов** — структура классов системы, их полей, методов интерфейсов и взаимосвязей.

Связи:



1. Ассоциация — связь, общий случай агрегации / композиции. Может именоваться
2. Наследование
3. Реализация — *клиент* реализует поведение, заданное *поставщиком*
4. Агрегация — ассоциация между целым и его частями (1 — контейнер другого)
5. Композиция — более строгая агрегация (по значению, а не ссылке). Зависимый класс уничтожается вместе с “контейнером”.
6. Зависимость — изменение 1ого влечет изменение другого (обратное неверно)

0	No instances (rare)
0..1	No instances, or one instance
1	Exactly one instance
1..1	Exactly one instance
0..*	Zero or more instances
*	Zero or more instances
1..*	One or more instances

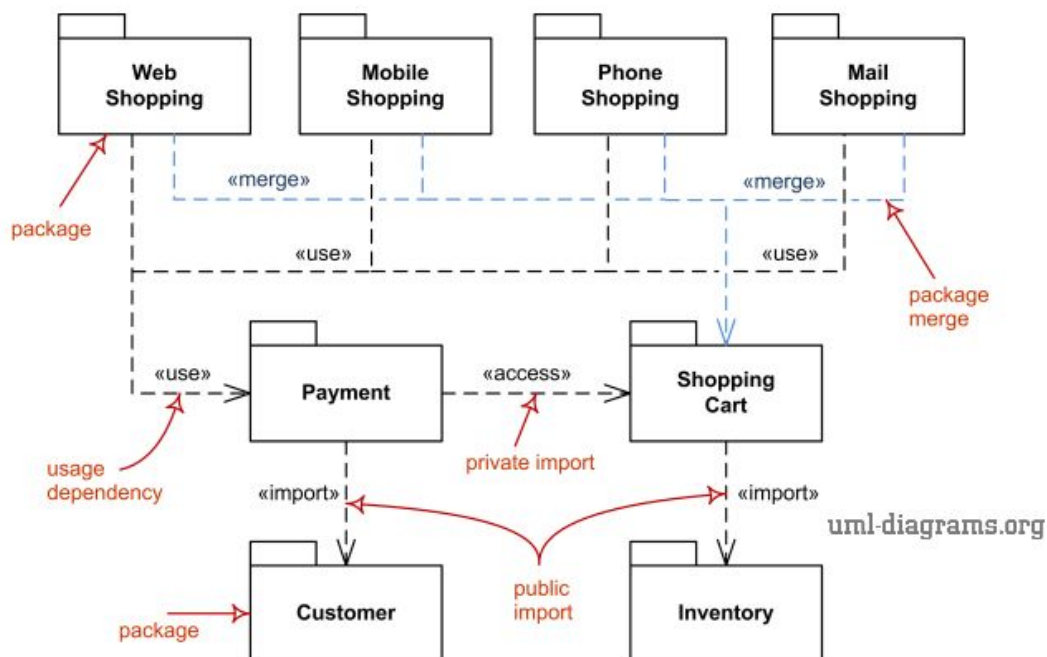
+	Public
-	Private
#	Protected
~	Package

В теории по диаграммам классов можно пытаться генерировать код.



## 11. Диаграммы объектов, диаграммы пакетов UML.

**Диаграмма пакетов** — визуализация зависимостей между пакетами. Как правило, используются для описания зависимостей между высокоуровневыми элементами больших систем. Пакет — пакеты в Java, пространство имен в C++.



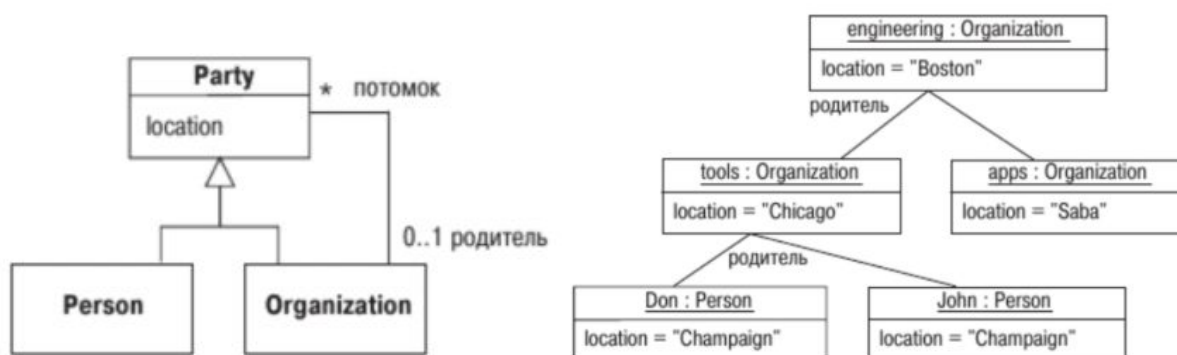
Связи:

1. Package *import* — добавление функциональности из пакета в namespace
2. Package *merge* — комбинирование namespace-ов
3. Package *use* — зависимость, необходимость в наличии реализации / определении
4. Package *access*

**Диаграмма объектов** — демонстрация моделируемых объектов и связей между ними *в определенный момент времени работы системы*. Иначе — визуализация объектов в памяти во время работы программы и их отношений.

Применение

1. Упрощенная визуализация диаграммы классов
2. Взаимосвязи объектов в предметной области



## 12. Диаграммы компонентов, диаграммы развертывания UML.

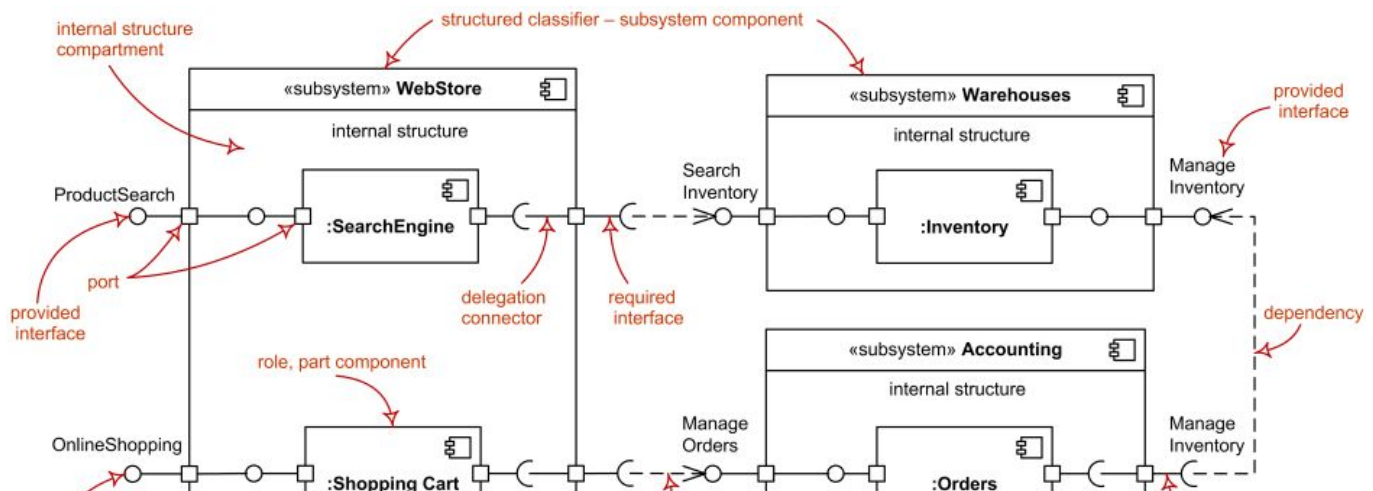
**Диаграмма компонентов** — визуализация компонентов системы и интерфейсов, их связывающих. Компонент — часть системы, инкапсулирующая определенное состояние и поведение.

Компоненты бывают

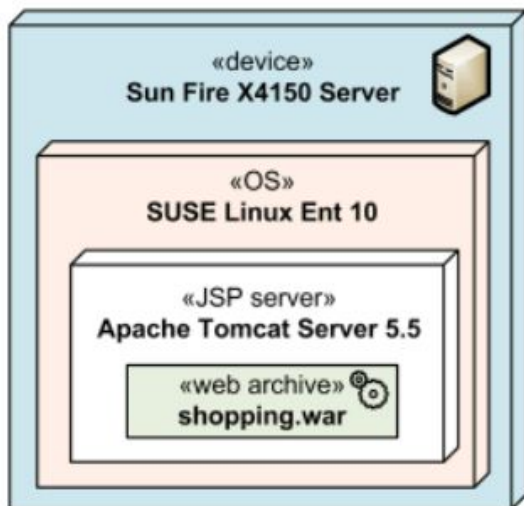
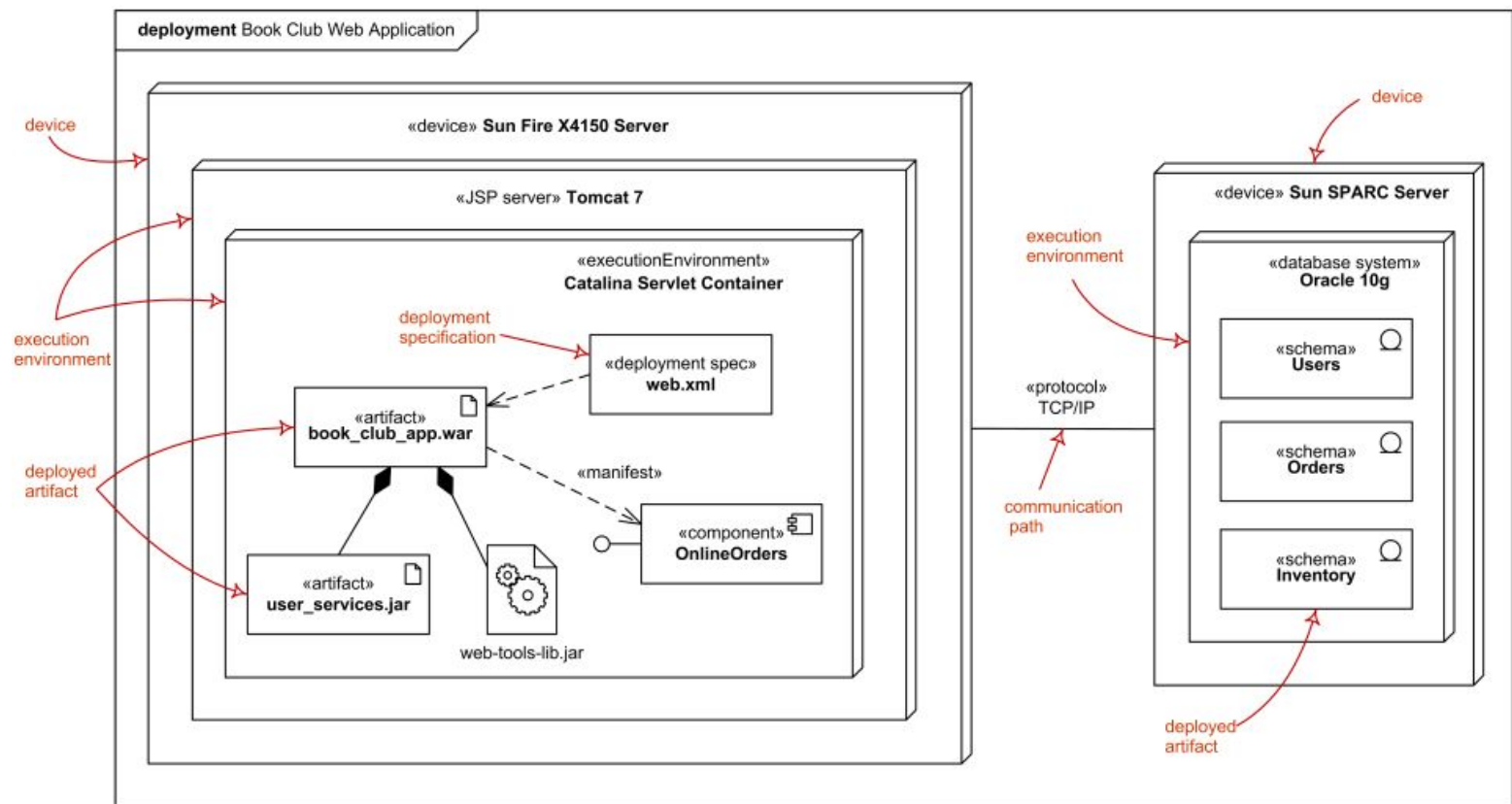
1. Логические
2. Физические

Диаграмма компонентов состоит из

1. *Component*
2. *Provided interface*
3. *Required interface*
4. *Port*
5. *Connectors*



**Диаграмма развертывания** — размещение компонентов (логических элементов системы) на физических или виртуальных устройствах.



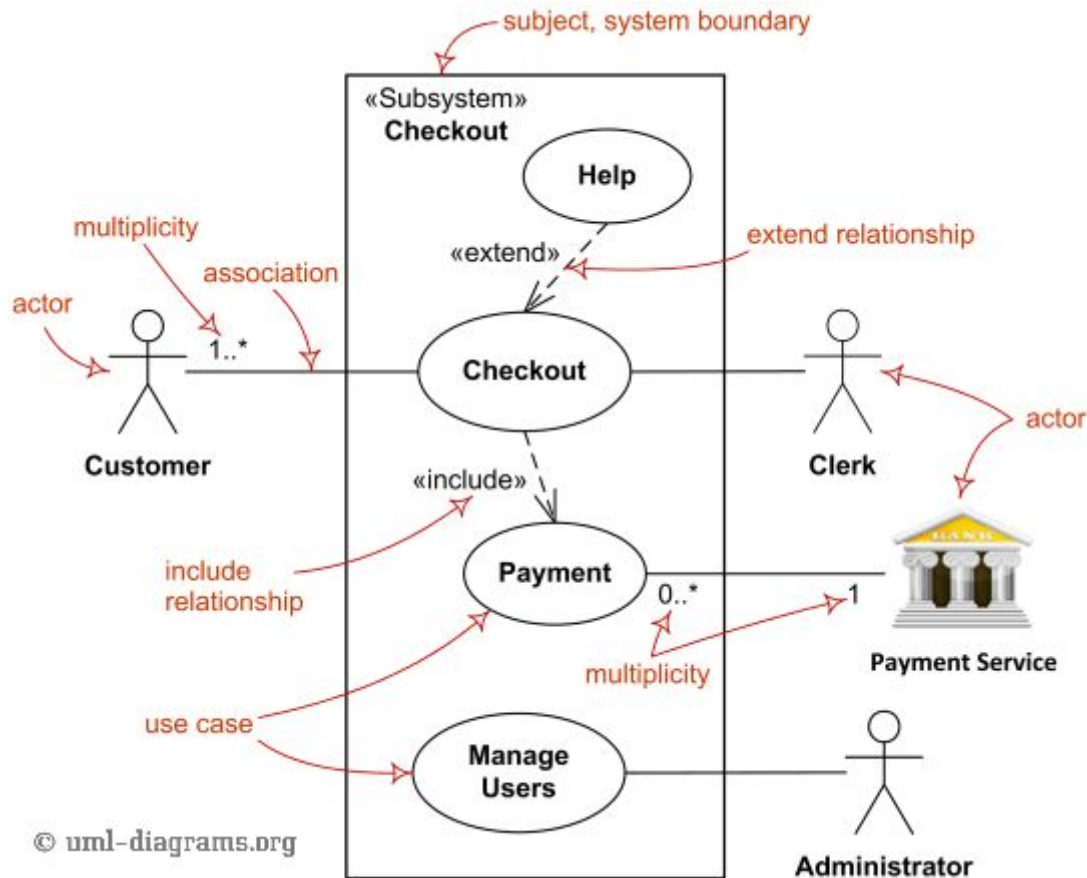
Параллелепипеды (вложенные) — устройства. В них располагаются артефакты (компоненты, бинарники).

Связи между устройствами — каналы связи (как правило через определенные протоколы).

Используются не только в конце разработки. Возможно такое, что физические устройства имеются заранее. Например, у заказчика уже есть сервера и под них нужно что-то написать.

## 13. Диаграмма случаев использования UML.

**Диаграмма случаев использования** — описывает набор действий, случаев использования, прецедентов (*actions*), которые должна реализовывать система (*subject, system boundary*) для внешнего пользователя (*actor*).



Связи:

1. Actor-Action — ассоциация
2. Action-Action — расширение, либо включение

Прецедент (*Action*) — цель использования системы пользователем.

**Use case** — список пошаговых событий, определяющих порядок взаимодействия между actor'ом и системой для достижения некоторой цели actor'a.

**Extend** — указывает на возможность особенного использования базового варианта (стрелки идут к базовому варианту от специальных). Возможно писать условие над стрелкой.

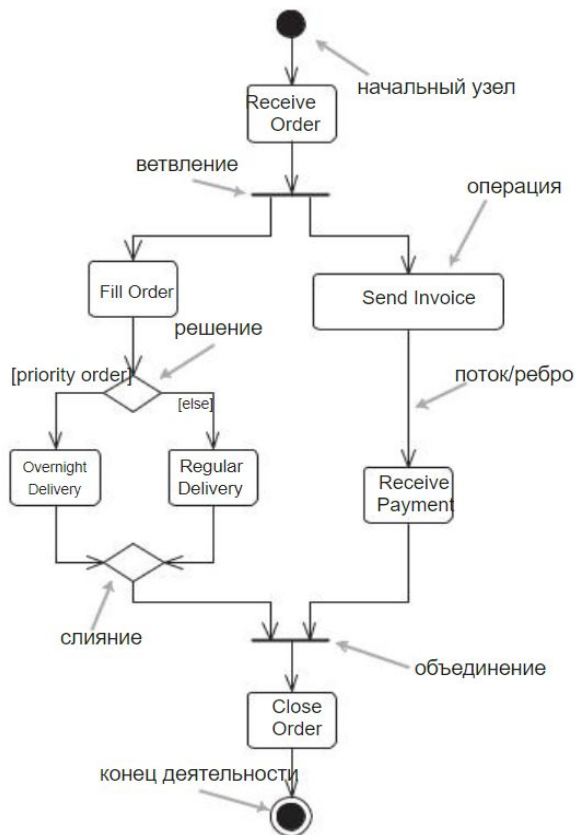
**Include** — иллюстрирует, какие сценарии использует базовый сценарий для выполнения операции (стрелки идут от базового варианта).

Но диаграммы слишком краткие, поэтому их нужно сопровождать текстовыми сценариями использования (для каждого случая использования). Типичная структура:

1. Заголовок — цель использования
2. Заинтересованные лица — все роли участников (ключевая — кто инициирует)
3. Предусловия — что должно быть перед исполнением сценария
4. Триггеры — начало исполнения сценария
5. Порядок событий
6. Альтернативные пути — расширения, реакции на ошибки
7. Постусловия — условия, которые должны быть выполнены после исполнения сценария

## 14. Диаграмма активностей UML.

**Диаграмма активностей** — описывает последовательный процесс выполнения определенной задачи. Используются для моделирования бизнес-процессов.



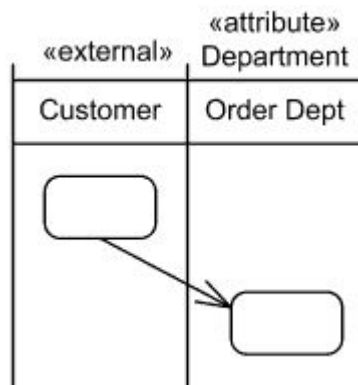
Узлы:

1. Action
2. Object
3. Control
  - a. Ветвление
  - b. Объединение (ждем все потоки)
  - c. Решение
    - i. покрывает все
    - ii. взаимоисключающие условия
  - d. Слияние
4. Начало / конец

Если один поток доходит до конца, завершаем работу.

Могут быть

1. Вложенные диаграммы (по сути функции)
2. Вызовы методов (этих функций)
3. Разделения на группы схожих действий:



<https://www.uml-diagrams.org/activity-diagrams.html>

<https://studfile.net/preview/6354103/page:34/>

## 15. Диаграммы конечных автоматов UML.

**Диаграмма конечных автоматов** — описывает поведение моделируемой системы через ее состояния и переходы между ними. Имеют исполнимую семантику => можно генерировать код.

Узлы:

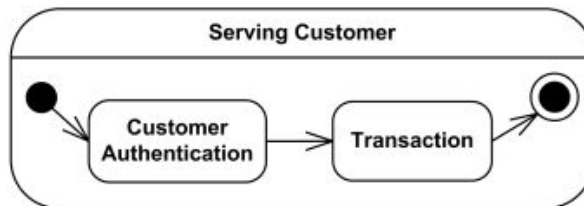
### 1. State

a. Simple state. Можно описывать поведение, которое будет выполняться:

- i. При входе в состояние
- ii. В процессе нахождения в состоянии
- iii. При выходе



b. Composite state. Содержит внутри несколько состояний



c. Submachine state. Ссылка на другой автомат

### 2. Pseudostate

- a. Initial / Terminate Pseudostate
- b. Entry / Exit Point
- c. Choice
- d. Fork
- e. Join

### 3. Параллельные состояния и History Pseudostate

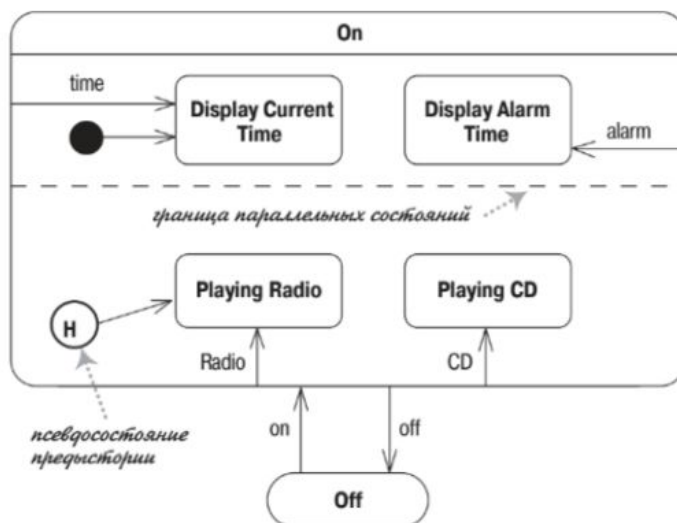
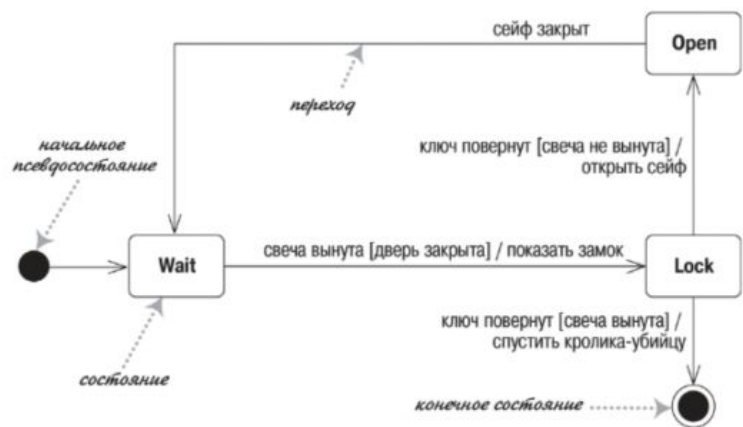


Таблица состояний — список событий и процедур их применения. Их можно использовать при генерации кода (произошло событие → смотрим текущее состояние → идем в таблицу → проверяем условия, выполняем действия выхода → переходим).



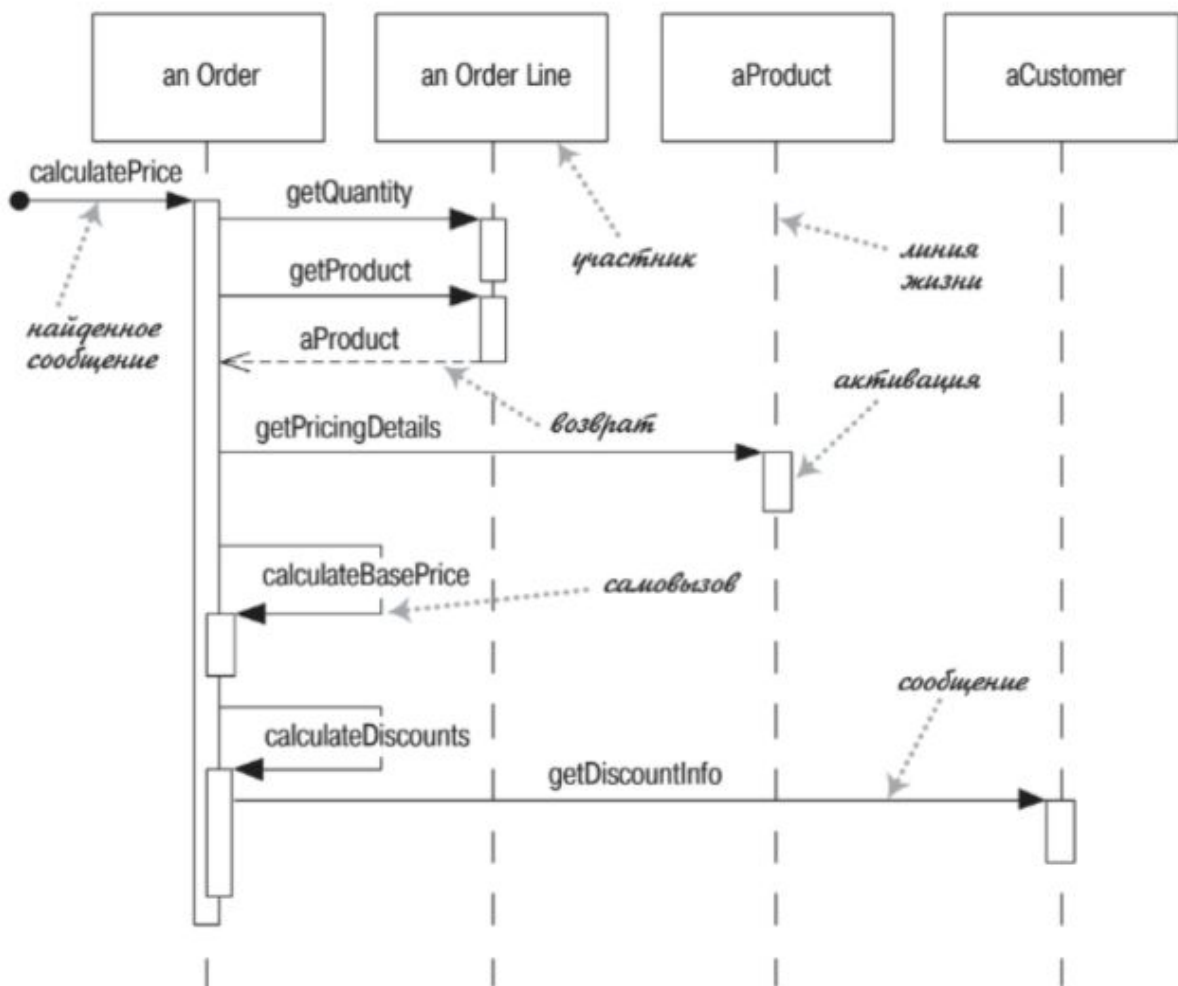
Исходное состояние	Целевое состояние	Событие	Защита	Процедура
Wait	Lock	Candle removed (свеча удалена)	Door open (дверца открыта)	Reveal lock (показать замок)
Lock	Open	Key turned (ключ повернут)	Candle in (свеча на месте)	Open safe (открыть сейф)
Lock	Final	Key turned (ключ повернут)	Candle out (свеча удалена)	Release killer rabbit (освободить убийцу-кролика)
Open	Wait	Safe closed (сейф закрыт)		



## 16. Диаграммы последовательностей UML.

**Диаграмма последовательностей** — визуализирует взаимодействие между объектами моделируемой предметной области. Фокус на сообщениях, которыми они обмениваются.

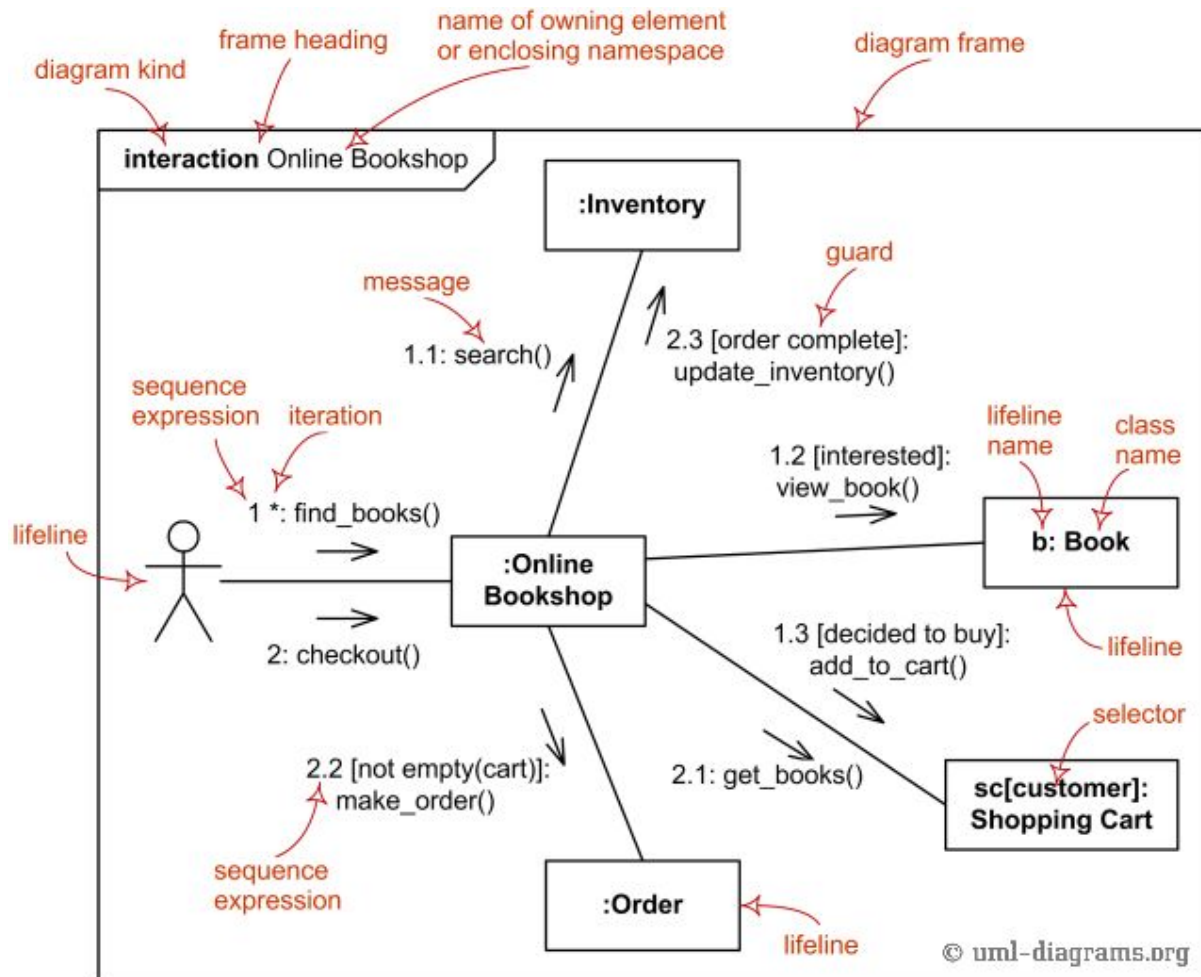
- Удобно для асинхронных вызовов.
- Применяются для
  - анализа предметной области
  - составления плана тестирования
  - визуализации логов системы



1. Lifeline — объект, участник обмена сообщениями. Соответствует **одной** сущности, участвующей в обмене сообщениями. Линия — время жизни объекта в системе, прямоугольники — время его активации.
2. Gate (*найденное сообщение*) — сообщение, покидающее или входящее в моделируемую систему (что-то типа интерфейса, только сообщение)
3. Messages — стрелки — информация, передающаяся при взаимодействии
4. Execution — прямоугольник — представляет собой процесс выполнения назначенного поведения. “Принимает” обычную стрелку, отправляет пунктирную.

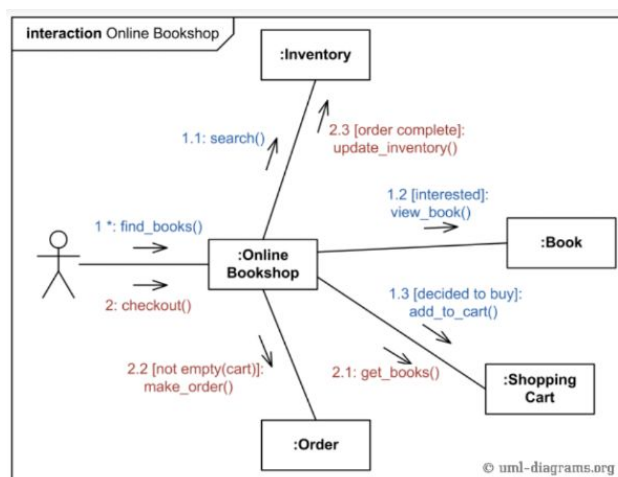
## 17. Диаграммы коммуникации UML.

**Диаграмма коммуникации** — визуализирует взаимодействие между объектами моделируемой системы, которое происходит за счет обмена сообщениями. Является простейшей диаграммой последовательностей (может быть к ней приведена или заменена).



Оперирует:

1. Frame
2. Lifeline — объект, участник обмена сообщениями. Соответствует **одной** сущности, участвующей в обмене сообщениями.
3. Message — передаваемая информация / сообщение.
  - а. Последовательность действий задается номерами:



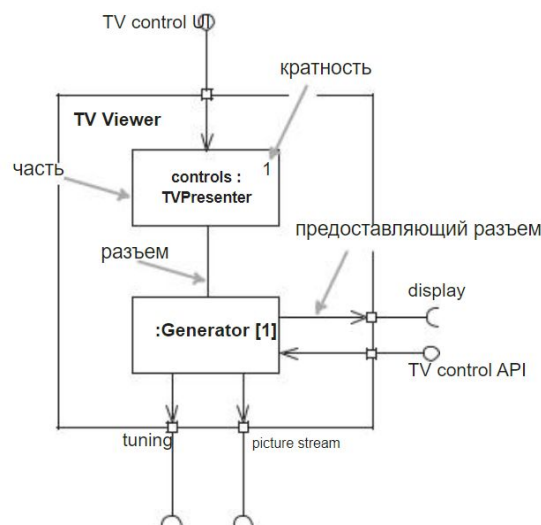
## 18. Диаграммы составных структур, коопераций, временные диаграммы.

### Диаграмма составных структур —

продвинутое диаграммы компонентов, но в качестве элементов не компоненты, а части (роли). В отличие от диаграммы компонентов фокус на runtime, а не этапе компиляции.

Если диаграммы компонентов — это диаграммы, показывающие структуру времени компиляции, то на диаграммах составных структур внутри объемлющей компоненты не другие компоненты, а роли.

Роль — это что-то вроде объекта, на диаграмме может быть несколько ролей одного типа, которые по-разному связаны друг с другом и делают разную работу. Так же, как объекты, роли имеют тип. В работающей системе роль может играть любая компонента правильного типа.

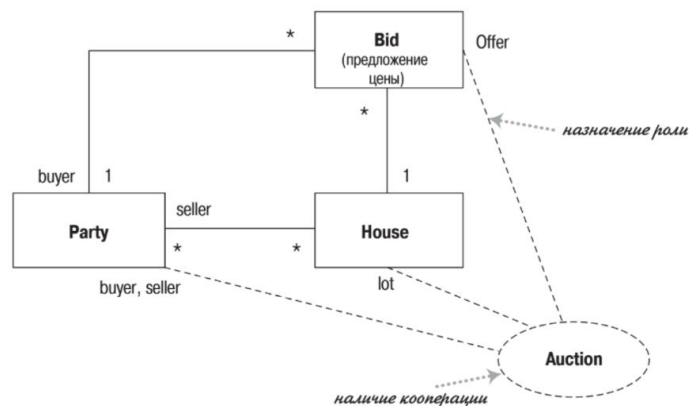


**Диаграмма коопераций** — показывают взаимодействие между объектами (ролями) в рамках одного сценария использования. Предоставляют способ группирования элементов взаимодействия, когда роли исполняются различными классами.

Диаграммы коопераций — это что-то среднее между диаграммами объектов и диаграммами классов. Вместо классов и объектов на них рисуются роли — сущности, на место которых может быть подставлен настоящий объект.

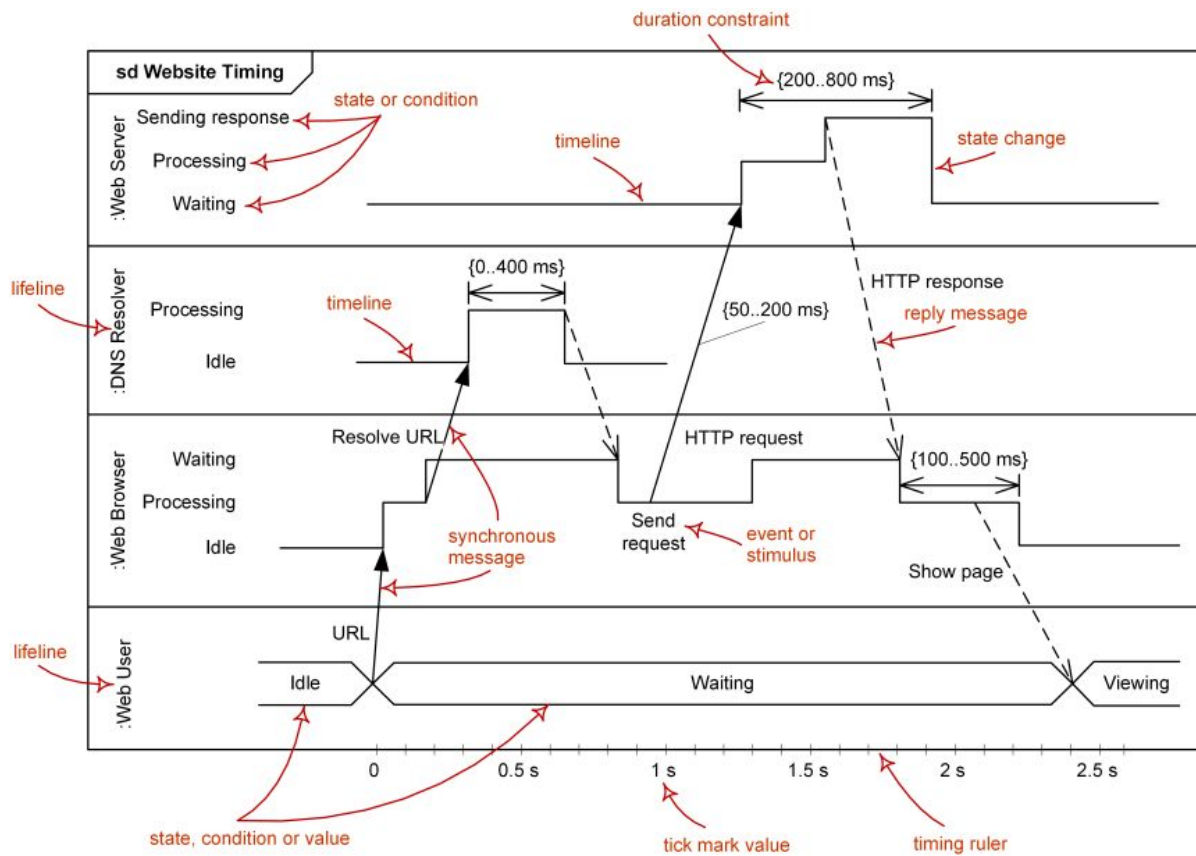
Нужны для иллюстрации одного сценария.

В UML 2 являются частью диаграмм составных структур.



**Временные диаграммы** — для моделирования временных ограничений в системах реального времени.

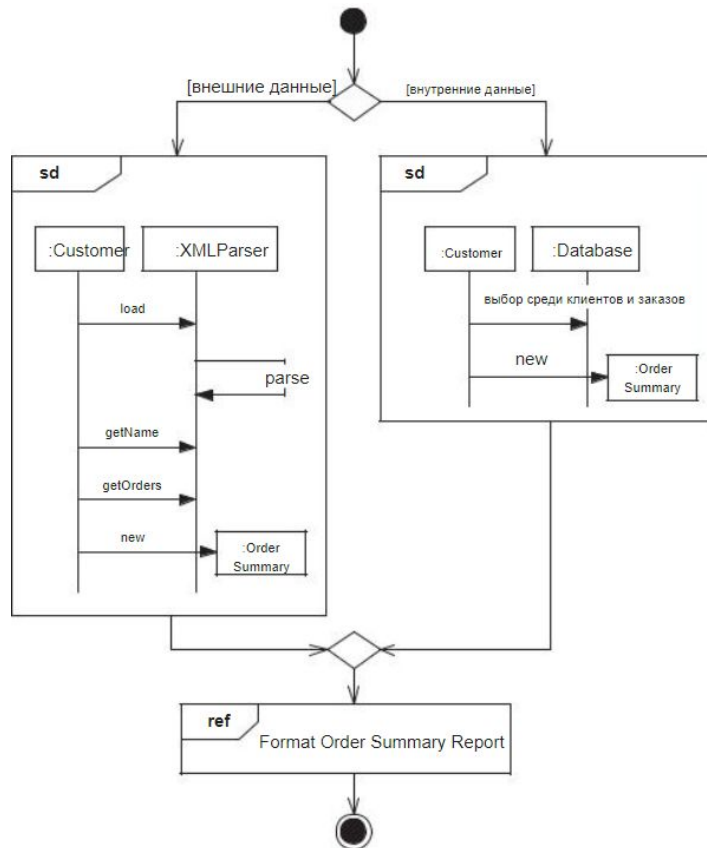
Есть временная шкала и объекты, которые могут находиться в разных состояниях (по вертикали слева). Переключение — скачок линии, либо (нижний блок на картинке) пересечение линий с обозначением состояния:

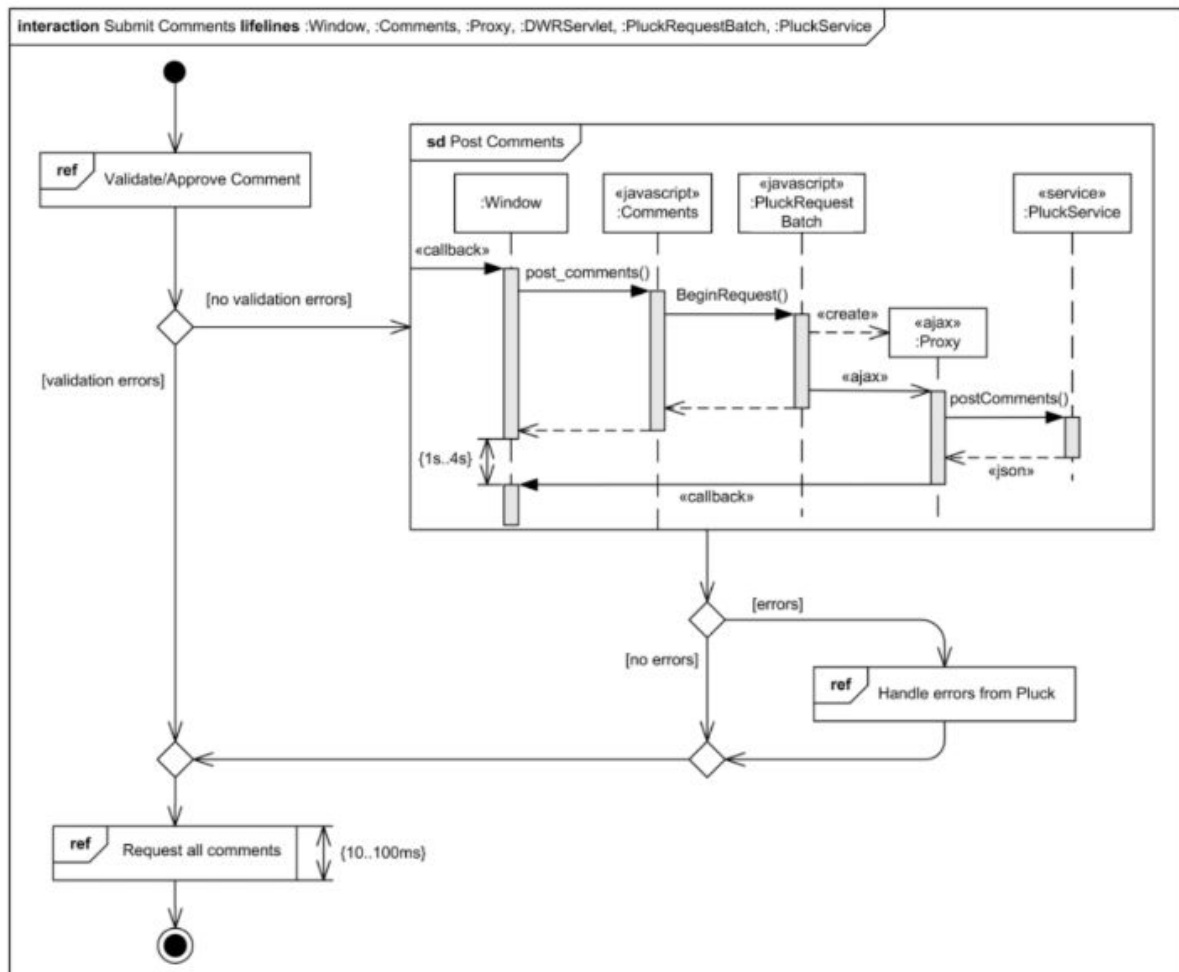


## 19. Диаграммы обзора взаимодействия, диаграммы потоков данных.

**Диаграмма обзора взаимодействия** — на базовом уровне диаграмма активностей, в которой в узлах вместо объектов и действий диаграммы последовательностей.

Применяются при сложном взаимодействии, когда диаграмма последовательностей неудобна. Например, сложный алгоритм, где куча ветвлений и циклов.



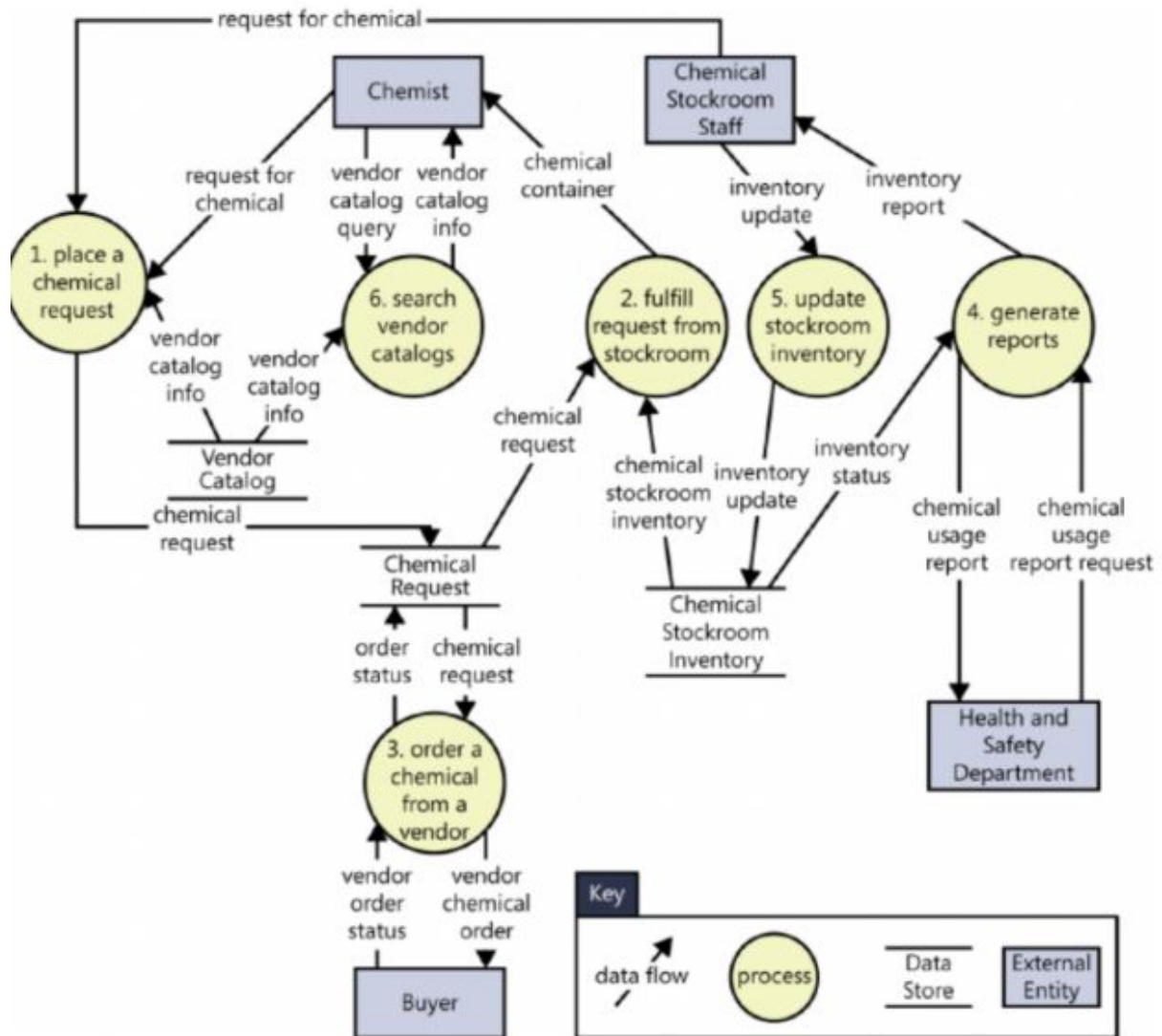


© <http://www.uml-diagrams.org/>

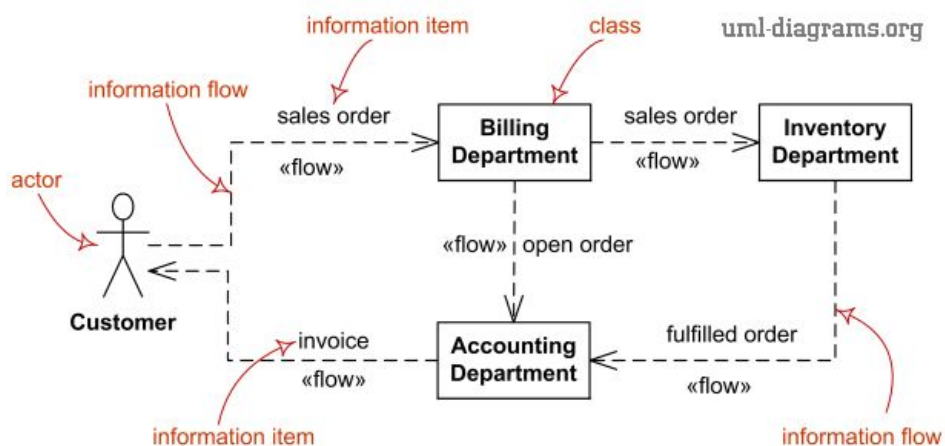
**Диаграмма потоков данных** — визуализирует обмен информацией между абстрактными сущностями системы. Литвинов говорит, что не является UML.

Синтаксис очень прост, есть один вид стрелок, который показывает поток данных, и три вида сущностей, между которыми, собственно, ходят данные:

- процесс (круг) — то, что может как-то преобразовывать данные внутри проектируемой системы
- внешняя сущность (прямоугольник) — то, что поставляет или потребляет данные
- хранилище (две горизонтальные линии) — то, где данные могут лежать, куда их можно поместить и забрать при необходимости



Information flow. Аналог из UML:



## 20. Диаграммы “Сущность-связь”.

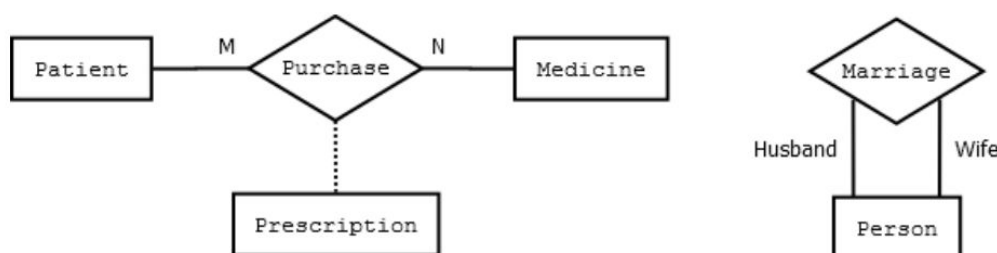
**Диаграмма сущность-связь** — визуализация модели Entity Relationship — формальная модель описания предметной области. Анализ данных.

**Сущность** — абстракция предметной области — прямоугольники

1. Независимо существует
2. Уникально идентифицируется (по ключевым атрибутам)
3. Однозначно интерпретируется

**Атрибуты** — уникально именованные характеристики сущностей — овалы.

**Связи** между сущностями — ромбы:



Кратность — количество участников с обоих концов. Как правило, нужна если хотим указать в качестве атрибута сущности другую сущность (ее ключ). Множество чисел на некоторых (или всех) концах связи

Необязательность (---) — некоторые из сторон связи могут иметь значение кратности 0.

Полнота (====) — покрывает ли связь все экземпляры сущности некоторых из entity-type. или жирная линия

Зависимые сущности — не могут существовать без связи с другими сущностями.



Ассоциативные сущности — отражают связи M:N.



По факту чаще используется нотация “вороньей лапки”: таблицы (сущности), поля в таблицах (атрибуты, ключи, уникальность...), связи между таблицами.



## 21. Понятие архитектурного стиля, трехзвенная архитектура

Так или иначе, большая часть информации взята из книги [Software Architecture: Foundations, Theory, Practice](#).

**Архитектурный стиль** — набор решений, которые:

1. применимы в выбранном контексте разработки
2. задают ограничения на принимаемые архитектурные решения, специфичные для определенных систем в этом контексте
3. приводят к желаемым положительным качествам получаемой системы

**Архитектурный стиль** — именованная коллекция архитектурных решений. *Менее узкоспециализированные, чем архитектурные паттерны.*

При этом:

- Одна система может включать в себя несколько архитектурных стилей
- Понятие стиля применимо и к подсистемам

Чем характеризуются стили:

- Набор используемых элементов архитектуры, то есть типы компонентов и соединителей, элементы данных (например, объекты, фильтры, сервера и т.д.)
- Набор правил конфигурирования
- “Топологические” ограничения на соединение элементов (например, компонент может быть соединён с максимум двумя компонентами)
- Семантика, стоящая за элементами

Чем хороши:

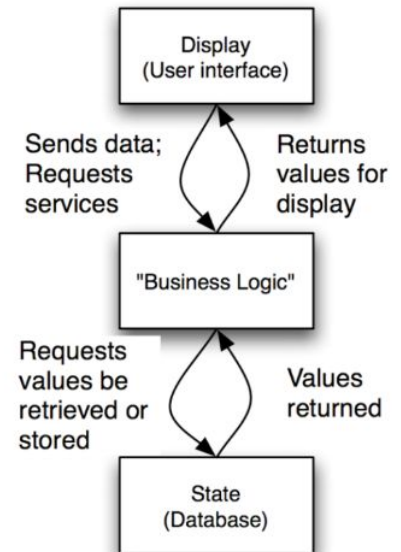
- Переиспользование архитектуры (для новых задач можно применять хорошо известные и изученные решения)
- Переиспользование кода (неизменяемые части, которые можно один раз реализовать)
- Упрощение общения и понимания системы
- Упрощение интеграции приложений
- Специфичные для стиля методы анализа
  - Возможны благодаря ограничениям на структуру системы
- Специфичные для стиля методы визуализации

**Архитектурный шаблон** — именованный набор ключевых проектных решений по эффективной организации подсистем, *применимых для повторяемых технических задач* проектирования в различных контекстах и предметных областях.

Трехзвенная архитектура — архитектурный шаблон (!):

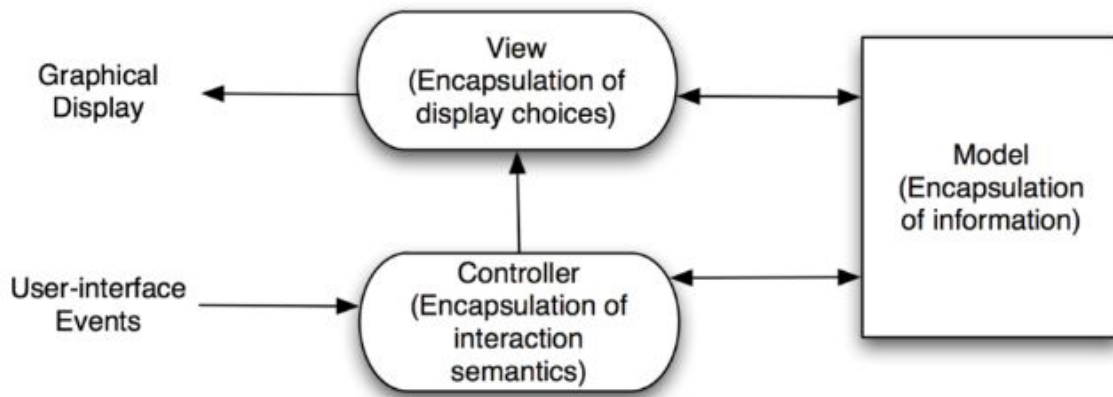
- Есть три уровня: уровень клиента, уровень бизнес-логики, уровень данных.

- **Уровень клиента:** интерфейс, предоставляемый пользователю. Не нагружен бизнес-логикой, не имеет прямой связи с базой данных. Сюда может быть вынесена простейшая логика вроде авторизации, алгоритмов шифрования, проверки данных на допустимость и т.д.
- **Уровень бизнес-логики:** сюда выносятся практически вся бизнес-логика (за исключением оговоренной логики на клиенте и логики в БД, например, хранимых процедур).
- **Уровень данных:** отвечает за хранение данных, как правило, посредством СУБД.



## 22. Model-View-Controller, Sense-Compute-Control.

**Model-View-Controller** — архитектурный шаблон, часто используемый при проектировании пользовательских интерфейсов. Разделяет данные, представление и взаимодействие с пользователем.



Есть следующие компоненты:

- **Model** — центральная компонента. Отражает внутреннюю организацию данных в программе, независимую от пользовательского интерфейса. Работает с данными, логикой приложения по внутренним правилам.
- **Controller** — считывает ввод пользователя и преобразует его в команды к модели или представлению (согласно Литвинову, естественное место для паттерна Command и Undo/Redo).
- **View** — любое представление информации (например, график или таблица). Возможно существование разных представлений (например, в виде графика для менеджмента и в виде таблиц для бухгалтерии).

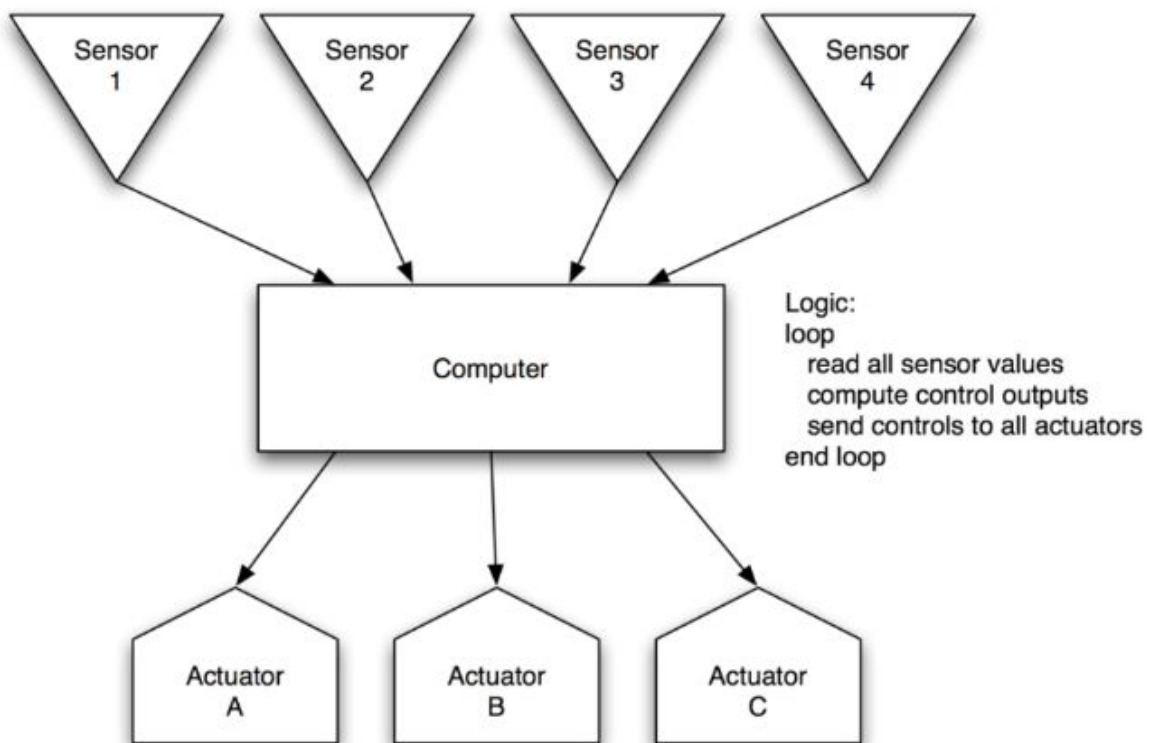
Взаимодействие происходит следующим образом:

- События о пользовательском действии поступают на контроллер.
- Контроллер может либо модифицировать данные в модели, либо само представление.
- Если данные в модели изменяются, она отправляет соответствующее событие представлению (либо контроллеру, если логика устроена таким образом).
- Представление в определенном формате показывает переданные из модели данные пользователю.

**Sense-Compute-Control** — архитектурный шаблон, применяющийся во встроенных системах и робототехнике.

Составные части:

- Сенсоры
- Процессор
- Приводы (двигатели)



В цикле считываем показания сенсоров, обрабатываем их и передаем на приводы.

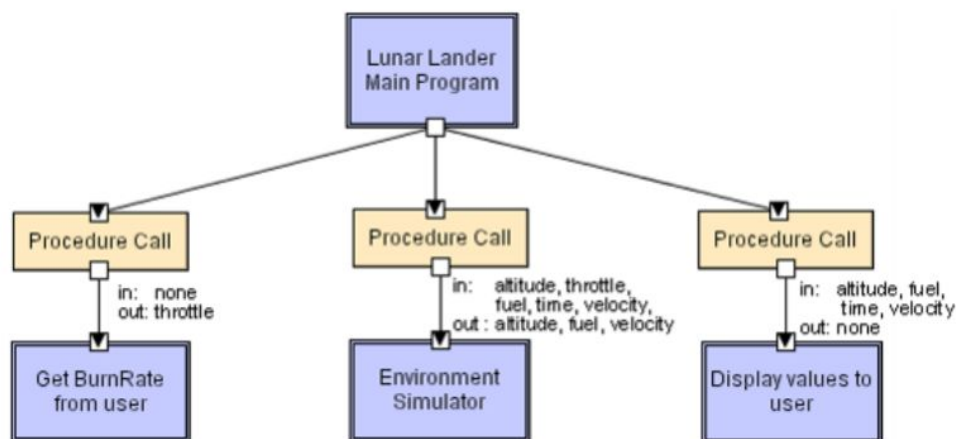
Примером может послужить программа для робота, едущего по линии:

1. Считываем показания с датчика цвета, чтобы узнать, насколько далеко мы от края линии.
2. Рассчитываем необходимое значение мощности двигателей для корректировки движения.
3. Передаем значения на моторы.

## 23. Структурный и объектно-ориентированный стили, слоистые архитектурные стили.

**Структурный стиль** — стиль, характерный для парадигмы структурного программирования. Программа представлена в виде иерархической структуры блоков (подпрограмм).

- **Компоненты:** главная программа и подпрограммы.
- **Коннекторы:** вызов функций/процедур.
- **Элементы данных:** передаваемые в процедуры и возвращаемые ими значения.
- **Топология:** иерархия вызовов.



© N. Medvidovic

**Объектно-ориентированный стиль** — стиль, характерный для объектно-ориентированной парадигмы программирования.

- **Компоненты:** объекты (состояния, строго инкапсулированные, с функциями для работы с ними).
- **Коннекторы:** вызовы методов.
- **Элементы данных:** аргументы, передаваемые методам.
- **Топология:** может различаться, включает иерархию наследования.

Важные положения:

- Компоненты программы являются объектами.
- Взаимодействие между объектами происходит посредством *вызовов методов*.
- Инварианты стиля:
  - Объекты отвечают за целостность своего внутреннего состояния

- Внутреннее представление скрыто от других объектов

Преимущества:

- + Декомпозиция системы в набор взаимодействующих агентов
- + Внутреннее представление объектов можно менять независимо
- + Близко к предметной области

Недостатки:

- Побочные эффекты при вызове методов
- Объекты вынуждены знать обо всех, от кого зависят

**Слоистый стиль** — архитектурный стиль, подразумевающий многоуровневую иерархическую организацию системы.

Каждый уровень является:

- **Сервером**: предоставляет свою функциональность слоям выше
- **Клиентом**: использует функциональность слоев ниже

Взаимодействие между слоями происходит посредством **протоколов**.

Примеры такой архитектуры: ОС (ядро, системные библиотеки, оболочка + утилиты), сетевые стеки протоколов (физический, канальный, сетевой, транспортный, прикладной уровни).

Преимущества:

- + Повышение уровня абстракции
- + Легкость в расширении
- + Изменения в каждом уровне затрагивают максимум два соседних
- + Возможны разные реализации уровня, если они удовлетворяют интерфейсу

Недостатки:

- Не всегда применим
- Проблемы с производительностью

**Клиент-сервер** — двухслойный стиль с доступом к сети.

- **Компоненты**: клиенты и сервер.
- **Коннекторы**: сетевые протоколы, удаленный вызов процедур.
- **Элементы данных**: значения, передаваемые по сети.
- **Топология**: двухслойная.

Инварианты:

- Серверы не знают ничего о клиентах, даже их количество
- Клиенты знают только про сервера и не могут общаться друг с другом

## 24. Пакетная обработка, каналы и фильтры, Blackboard.

**Пакетная обработка** — один из старейших стилей проектирования (“Прадедушка стилей”).

- **Компоненты:** независимые программы.
- **Коннекторы:** человек, вручную “переносящий ленты от программы к программе”.
- **Элементы данных:** стандартные для ОС форматы данных.
- **Топология:** линейная.

Представляет собой набор программ, которые:

- запускаются последовательно
- передают данные стандартным для ОС образом (файлы, пайплайны и т.д.)

Программы приходится запускать человеку (либо вручную, либо писать скрипт).

**Каналы и фильтры** — стиль проектирования, в котором программа представляет собой набор *фильтров* (компонента, преобразующая входной поток данных в выходной) и их соединителей — *каналов*.

- **Компоненты:** независимые программы/элементы программы — фильтры.
- **Коннекторы:** явно заданные потоки данных, сервисы ОС.
- **Элементы данных:** должны быть линейными потоками данных (что бы это ни значило).
- **Топология:** пайплайн.

Инварианты стиля:

- фильтры независимы от других фильтров или глобального состояния
- фильтры не знают о фильтрах до и после

Возможные вариации:

- **Конвейеры** — линейные последовательности фильтров
- **Ограниченные каналы** — где канал это очередь с ограниченным количеством элементов
- **Типизированные каналы** — где каналы отличаются по типу передаваемых данных

Преимущества:

- + Поведение системы — это просто последовательное применение поведений компонентов
- + Легко добавлять, заменять и переиспользовать фильтры
- + Любые два фильтра можно использовать вместе

- + Широкие возможности для анализа (пропускная способность, задержки, deadlock-и)
- + Широкие возможности для параллелизма

Недостатки:

- *Последовательное исполнение*
- Проблемы с интерактивными приложениями
- Пропускная способность определяется самым “узким” элементом

**Blackboard** — архитектурный стиль, подразумевающий наличие некоей центральной структуры данных (сам “Blackboard”) и взаимодействующих с ней компонентов (“Knowledge sources”).

- **Компоненты:** сам blackboard и независимые программы/части системы (“Knowledge sources”).
- **Коннекторы:** способ обращения к blackboard (прямой доступ к памяти, запросы к БД и т.п.).
- **Элементы данных:** данные в blackboard.
- **Топология:** звезда, в центре blackboard.

Инварианты стиля:

- Управление системой осуществляется только через состояние доски
- Компоненты не знают друг о друге и не имеют своего состояния

Стиль возник в задачах, связанных с искусственным интеллектом.



## 25. Стили с неявным вызовом, Publish-Subscribe.

**Стили с неявным вызовом** — архитектурные стили, которые подразумевают наличие оповещения о некотором событии вместо явного вызова метода.

Принцип работы:

- “Слушатели” могут подписаться на событие
- Система при наступлении события сама вызывает все зарегистрированные методы слушателей

Компоненты имеют два вида интерфейсов — *методы* и *события*. Взаимодействуют посредством *прямых вызовов* методов и *неявных вызовов* по наступлению некоторого события.

Инварианты стиля:

- Те, кто производит события, не знают, кто и как на них отреагирует
- Не делается никаких предположений о том, как событие будет обработано и будет ли вообще

Преимущества:

- Переиспользование компонентов (очень низкая связность между компонентами)
- Лёгкость в конфигурировании системы (как во время компиляции, так и во время выполнения)

Недостатки:

- Зачастую не интуитивная структура системы
- Компоненты не управляют последовательностью вычислений
- Непонятно, кто отреагирует на запрос и в каком порядке придут ответы
- Тяжело отлаживаться
- Гонки даже в однопоточном приложении

**Издатель-подписчик** — один из стилей с неявным вызовом.

- **Компоненты:** издатель, подписчики, всевозможные прокси для распределенного доступа.
- **Коннекторы:** вызов процедур (например, паттерн Наблюдатель)/сетевые протоколы.
- **Элементы данных:** подписки, нотификации, опубликованная информация.
- **Топология:** подписчики либо напрямую соединены с издателем, либо через посредников.

Подписчики регистрируются, чтобы получать нужные им сообщения или данные. Издатели публикуют сообщения, синхронно или асинхронно.

Преимущества: очень низкая связность между компонентами, при этом высокая эффективность распределения информации.

**Событийно-ориентированный подход** — стиль с неявным вызовом в котором независимые компоненты посылают (генерируют) и принимают (потребляют) события, передаваемые по шинам (событий).

- **Компоненты:** независимые генераторы/потребители, работающие параллельно.
- **Коннекторы:** шина событий (возможно не одна).
- **Элементы данных:** события (*first-class citizen*).
- **Топология:** компоненты общаются только с шинами событий, не друг с другом.

Преимущества: легкость масштабирования и добавления новой функциональности, эффективно для распределенных приложений.

## 26. Peer-to-peer, C2, CORBA.

**Peer-to-Peer** — стиль проектирования, в котором состояние и поведение распределены между компонентами, которые могут выступать как клиенты и как серверы.

Взаимодействие происходит, как правило, посредством сетевых протоколов.

- **Компоненты:** независимые собеседники (peers).
- **Коннекторы:** сетевые протоколы, часто уникальные.
- **Элементы данных:** сетевые сообщения.
- **Топология:** связь всех со всеми.

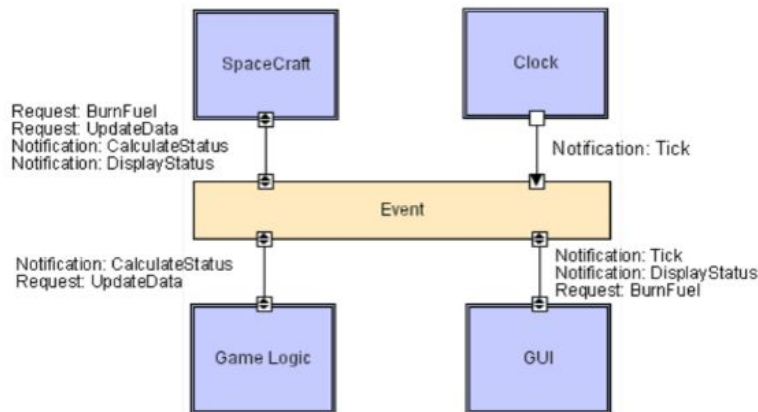
Преимущества:

- Хорош для распределённых вычислений
- Устойчив к отказам
- Если протокол взаимодействия позволяет, легко масштабируется

**Гетерогенные стили** — сложные стили, полученные соединением простых (например, REST, C2, CORBA и прочие).

**C2 (components and connectors)** — стиль, выросший из желания получить преимущества шаблона MVC в распределенной, гетерогенной системе. Представляет собой стиль с неявным вызовом, где компоненты общаются только через коннекторы, маршрутизирующие сообщения.

- **Компоненты:** независимые, потенциально параллельные производители или потребители.
- **Соединители:** маршрутизаторы сообщений (коннекторы), которые могут фильтровать, преобразовывать и рассылать сообщения двух видов: нотификации и запросы.
- **Топология:** у каждого коннектора и компоненты есть *top* и *bottom*. *Top* одной компоненты может быть соединен с *bottom* ровно одного (!) коннектора, а *bottom* может быть соединен с *top* ровно одного коннектора. Также могут быть соединены *top* и *bottom* коннекторов (тут уже нет ограничения на единственность). Никакие другие соединения недопустимы. Таким образом, многие компоненты могут быть соединены с одним коннектором. Получается иерархическая структура.
- **Элементы данных:** сообщения, содержащие данные
  - Нотификации анонсируют изменения в состоянии (передаются “вниз” по системе).
  - Запросы — запрашивают выполнение действия (передаются “вверх” по системе).



**CORBA** — стиль для архитектуры middleware (связующее программное обеспечение, по сути инфраструктура), поддерживающий разработку приложений на основе распределенных объектов.

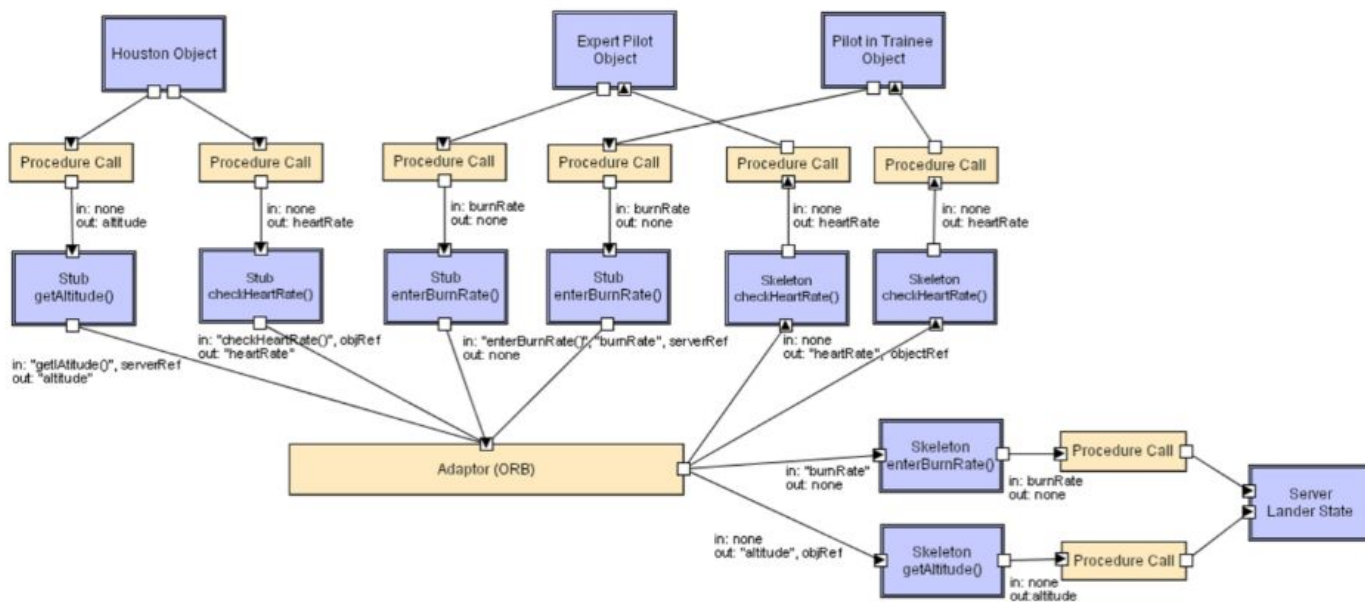
Главная идея — разбить приложение на объекты (возможно распределенные, реализованные на разных языках), которые предоставляют один или несколько интерфейсов, описанных в платформо независимой строго типизированной нотации (называется Interface Definition Language или **IDL**). Все взаимодействие происходит с помощью удаленных вызовов процедур (**RPC**) этих интерфейсов.

- **Компоненты:** распределенные объекты с четко описанным представляемым интерфейсом.
- **Коннекторы:** удаленный вызов процедур.
- **Элементы данных:** аргументы методов, возвращаемые значения, исключения.
- **Топология:** граф объектов в самом общем смысле.

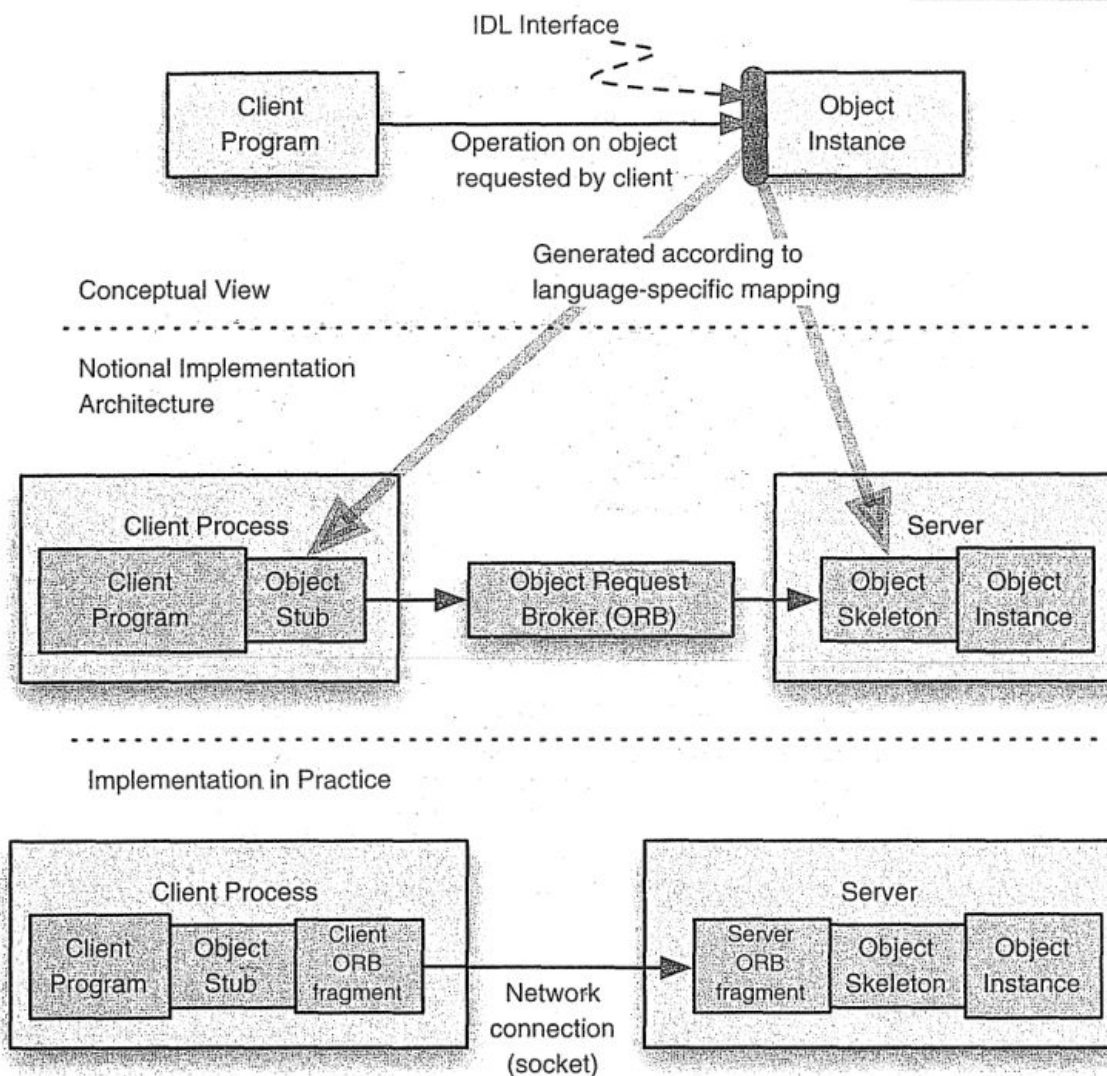
Дополнительные ограничения:

- Передаваемые при вызове метода данные должны быть сериализуемы
- Вызывающие должны обрабатывать ошибки, связанные с работой сети

**Преимущества:** независимость от платформы, языка и местоположения сервиса.



© N. Medvidovic



## 27. Понятие Domain-Driven Design, единый язык, изоляция предметной области.

Про DDD Литвинов брал информацию из [книжки](#).

**Domain-Driven Design (или предметно-ориентированное проектирование)** — популярная методология проектирования ПО, которая основана на анализе предметной области и реализации модели предметной области в приложении.

Архитектуру в рамках этой методологии предлагается строить не исходя из сиюминутных потребностей реализации, а вокруг “*смыслового ядра*”, которое отражает основные сущности реального мира, с которым будет работать программа (или семейство программ).

### Единый язык системы:

Программисты и специалисты предметной области разговаривают на разных жаргонах, следовательно, нужен общий язык, чтобы наладить коммуникацию. В единый язык входят:

- термины предметной области — как правило, они становятся *именами классов*;
- отношения между понятиями и действия, которые сущности могут выполнять — это *имена методов*;
- *имена паттернов*, используемых при проектировании системы, даже если их в предметной области нет (например, “репозиторий” или “спецификация”);
- *понятия, относящиеся к высокоуровневой архитектуре системы, которые напрямую невыразимы в коде* — уровни, ограничения на взаимодействие (например, понятие “канал” в архитектурном стиле “каналы и фильтры” может явно в коде никак не выражаться, но в архитектуре играет ключевую роль).

Особенности единого языка:

- Единый язык не создаётся мгновенно, он эволюционирует вместе с пониманием предметной области, моделью и кодом, который эту модель реализует.
- Если проект состоит из нескольких более-менее обособленных частей, то каждая из них может иметь свой язык.

### Изоляция предметной области:

Основная идея DDD:

- Код модели — простой, небольшой по размеру, содержит только важные для смысла системы вещи.
- Всё остальное (пользовательский интерфейс, сеть, база данных и т.д.) — в отдельных модулях, использует классы доменной модели.

Самый популярный способ достижения такого разделения — **уровневая архитектура**. Программа реализуется в виде набора уровней, где каждый уровень может взаимодействовать только с уровнями ниже.

Способов разделения на уровни бывает много (трехзвенная архитектура, семь уровней OSI, четыре уровня TCP/IP и т.д.), DDD требует лишь, чтобы среди всех уровней был **уровень предметной области**, на котором и сосредоточены все классы модели.

Типичный способ разделения:



1. **Уровень интерфейса** — отвечает только за взаимодействие с пользователем и реализует только отображение и обработку событий (никакой содержательной логики).
2. **Операционный уровень** — занимается координацией действий бизнес-объектов, которые находятся на уровне предметной области, тоже должен быть очень простым. Его ответственность — это инициализировать систему и по сигналу от интерфейса инициировать нужные пользователю операции.
3. **Уровень предметной области** — содержит бизнес-объекты, которые ничего интересного кроме, собственно, реализации бизнес-правил, не делают.
4. **Инфраструктурный уровень** — содержит все вспомогательные вещи, не специфичные для данной предметной области, например, код работы с базой данных или код работы с сетью.





## 28. Основные структурные элементы модели предметной области.

### Ассоциации:

- В идеале почти все ассоциации должны быть *однонаправлены* (иначе возникает сильная связанность объектов).
- По возможности необходимо *минимизировать количество и множественность* ассоциаций (“страна – год – президент” вместо “страна – президент”). Меньше связей — проще анализ и сопровождение.

### Сущности и объекты:

- *Сущность (Entity)* — объект, обладающий собственной идентичностью
  - Нужна операция идентификации
  - Нужен способ поддержания идентичности
- *Объект-значение (Value object)* — объект, полностью определяемый своими атрибутами
  - “Лучше”, чем сущность
  - Как правило, немутабельны
  - Могут быть разделяемыми
- *Служба (Service)* — объект, представляющий операцию
  - Как правило, не имеет собственного состояния
  - Операции нет естественного места в других классах модели
- *Модуль (Module)* — смысловые части модели

Про сущности из книги:

*“Для некоторых объектов определение через атрибуты не является главным. Они представляют собой индивидуально существующие логические единицы (сущности), протяженные во времени и часто проходящие через несколько различных представлений. Иногда такой объект приходится ставить в соответствие другому объекту, хотя их атрибуты отличаются. Или же объект требуется отличать от другого объекта, хотя их атрибуты могут быть одинаковыми. Путаница в объектах может привести к искажению данных”.*

Про объекты-значения оттуда же:

*Если элемент модели полностью определяется своими атрибутами, то его следует считать ОБЪЕКТОМ-ЗНАЧЕНИЕМ. Сделайте так, чтобы он отражал смысл заложенных в него атрибутов и придайте ему соответствующую функциональность. Считайте такой объект неизменяющимся. Не давайте ему индивидуальности, вообще избегайте любых сложностей, неизбежных при программном управлении СУЩНОСТЯМИ.*

Про модули:

*Выберите такие МОДУЛИ, которые бы рассказывали историю системы и содержали связанные наборы понятий. От этого часто сама собой возникает низкая зависимость МОДУЛЕЙ друг от друга. Но если это не так, найдите способ изменить модель таким образом, чтобы отделить понятия друг от друга, или же поищите пропущенное в модели понятие, которое могло бы стать основой для МОДУЛЯ и тем самым свести элементы модели вместе естественным, осмысленным способом.*

*Добивайтесь низкой зависимости модулей друг от друга в том смысле, чтобы понятия в разных модулях можно было анализировать и воспринимать независимо друг от друга.*

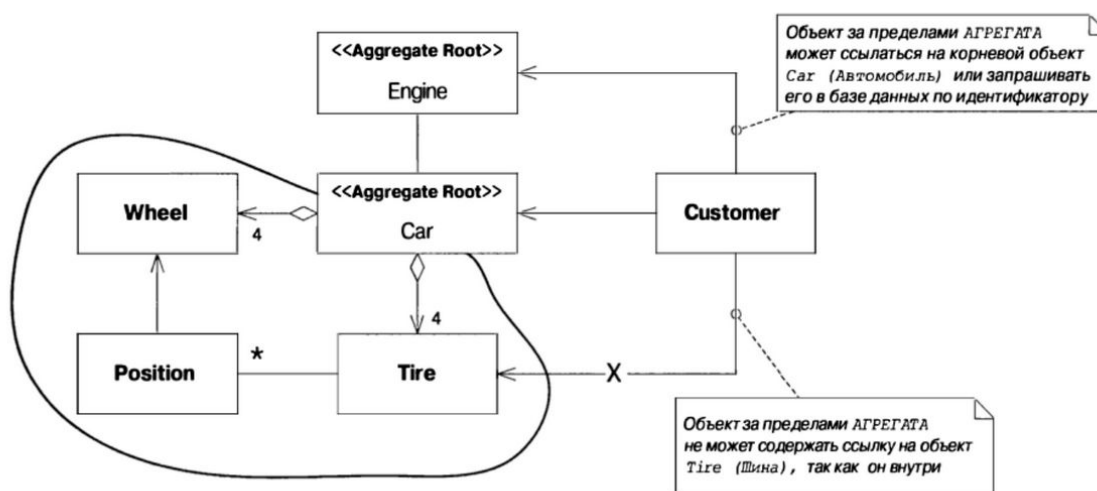
*Дорабатывайте модель до тех пор, пока в ней не возникнут естественные границы в соответствии с высокоуровневыми концепциями предметной области, а соответствующий код не разделится соответствующим образом.*

*Дайте модулям такие имена, которые войдут в ЕДИНЫЙ ЯЗЫК. Как сами МОДУЛИ, так и их имена должны отражать знание и понимание предметной области.*

## 29. Паттерны “Агрегат”, “Фабрика”, “Репозиторий”.

**Агрегат** — изолированный кусок модели, имеющий **корень** и **границу**

- *Корень* — глобально идентичный объект-сущность
- *Граница* — отделение агрегата от остальной системы
- Остальные объекты в агрегате идентичны локально
- Извне агрегата можно хранить ссылку только на корень
- Можно отдавать только временную ссылку на внутренности
- Корень отвечает за поддержание инвариантов всего агрегата
- Агрегат и его составляющие имеют одинаковое время жизни



Другой пример - заказ в магазине. Он содержит в себе товары, но к ним нельзя получить доступ в обход заказа. Заказ поддерживает инварианты - например, вес заказа не должен быть больше 10 кг.

**Фабрика** служит для создания объектов или агрегатов

- Скрывает внутреннее устройство конструируемого объекта
- Операция создания “атомарна” и обеспечивает инварианты
- Изолирует сложную операцию создания
- Как правило, не имеет бизнес-смысла, но является частью модели
- Реализуется аж несколькими разными паттернами
- В общем смысле, фабрика может использоваться для восстановления целого объекта из базы данных по идентификатору.

**Репозиторий (Хранилище)** хранит объекты и предоставляет к ним доступ (виртуальное множество объектов)

- Может инкапсулировать запросы к БД
- Может использовать фабрики
- Может обладать развитым интерфейсом запросов

Не обязательно именно хранит, но может также создавать и загружать требуемые объекты с помощью фабрик.

## 30. Моделирование ограничений, паттерн “Спецификация”.

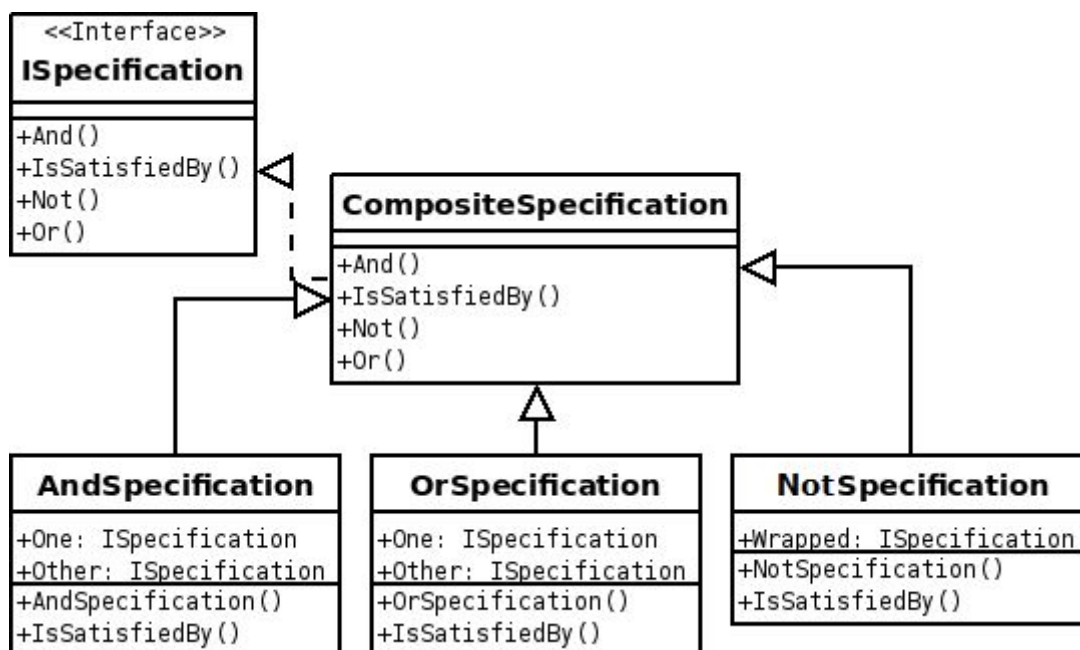
**Спецификация** инкапсулирует ограничение в отдельном объекте

- Предикат (Бизнес правило)
- Может быть использована для выборки или конструирования объектов (Отбор по череде условий, создание объектов, удовлетворяющих условиям)

Объекты класса содержат функцию “**IsSatisfiedBy(*Item*)**”, которая проверяет соответствие другого объекта некоторому условию.

### Композитная спецификация

Содержит цепочки простых спецификаций.



## 31. Паттерн “Компоновщик”.

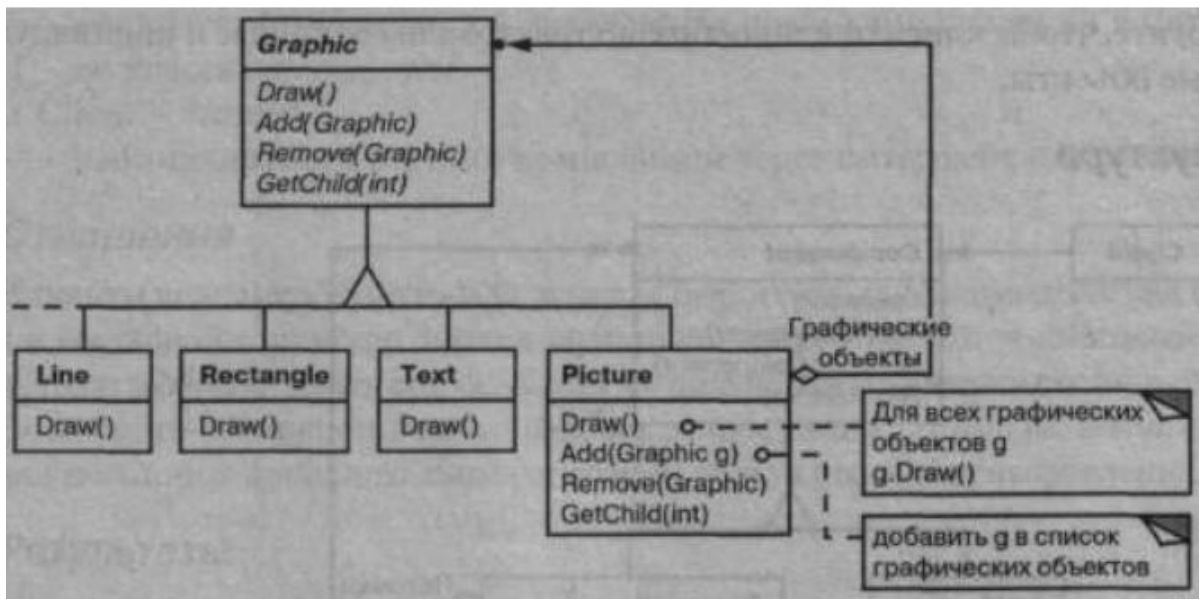
### “Компоновщик” (Composite)

Древовидная структура. Позволяет задать интерфейс, через который будет осуществляться единообразный доступ к простым и составным объектам.

- Представление иерархии объектов вида часть-целое
- Единообразная обработка простых и составных объектов
- Простота добавления новых компонентов
- Пример:
  - Синтаксические деревья

### Детали реализации

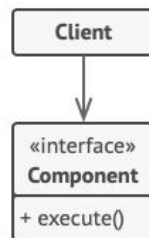
- Можно хранить ссылку на родителя
  - Может быть полезна для простоты обхода
  - “Цепочка обязанностей”
  - Но дополнительный инвариант
  - Обычно реализуется в классе поддерева
- Разделяемые под-деревья и листья
  - Позволяют сильно экономить память
  - Проблемы с навигацией к родителям и разделяемым состоянием
  - Паттерн “Приспособленец”
- Идеологические проблемы с операциями для работы с потомками
  - Не имеют смысла для листа
    - Можно считать Leaf Composite-ом, у которого всегда 0 потомков
  - Операции add и remove можно объявить не в интерфейсе, а в классе поддерева, тогда придётся делать cast
    - Иначе надо бросать исключения в add и remove
- Операция getComposite() – более аккуратный аналог cast-a
- Порядок потомков может быть важен, может нет
- Кеширование информации для обхода или поиска
  - Например, хранить цену поддерева
  - Инвалидация кеша
- Удаление потомков
  - Если нет сборки мусора, то лучше в Composite
  - Следует опасаться разделяемых листьев/поддеревьев



**1** Компонент определяет общий интерфейс для простых и составных компонентов дерева.

**2** Лист — это простой компонент дерева, не имеющий ответвлений.

Из-за того, что им некому больше передавать выполнение, классы листьев будут содержать большую часть полезного кода.



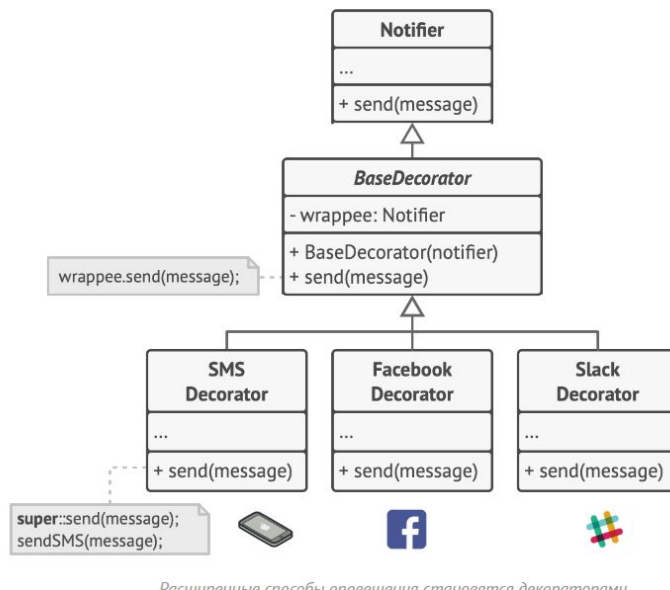
**4** Клиент работает с деревом через общий интерфейс компонентов.

Благодаря этому, клиенту не важно, что перед ним находится — простой или составной компонент дерева.

**3** Контейнер (или композит) — это составной компонент дерева. Он содержит набор дочерних компонентов, но ничего не знает об их типах. Это могут быть как простые компоненты-листья, так и другие компоненты-контейнеры. Но это не является проблемой, если все дочерние компоненты следуют единому интерфейсу.

Методы контейнера переадресуют основную работу своим дочерним компонентам, хотя и могут добавлять что-то своё к результату.

## 32. Паттерн “Декоратор”.



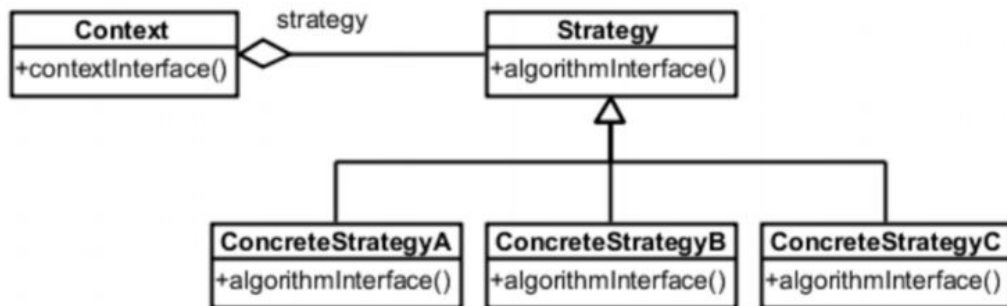
**Декоратор** позволяет оборачивать объекты, чтобы добавлять композитную функциональность. Декоратор имеет интерфейс оборачиваемого объекта и в простейшем случае просто пробрасывает запрос в него. Декораторы могут объединяться в цепочки.

- Динамическое добавление (и удаление) обязанностей объектов
- Большая гибкость, чем у наследования
- Позволяет избежать перегруженных функциональностью базовых классов
- Много мелких объектов
- Интерфейс декоратора должен соответствовать интерфейсу декорируемого объекта
  - Иначе получится “Адаптер”
- Если конкретный декоратор один, абстрактный класс можно не делать
- Component должен быть по возможности небольшим (в идеале, интерфейсом)
  - Иначе лучше паттерн “Стратегия”
  - Или самодельный аналог, например, список “расширений”, которые вызываются декорируемым объектом вручную перед операцией или после нее

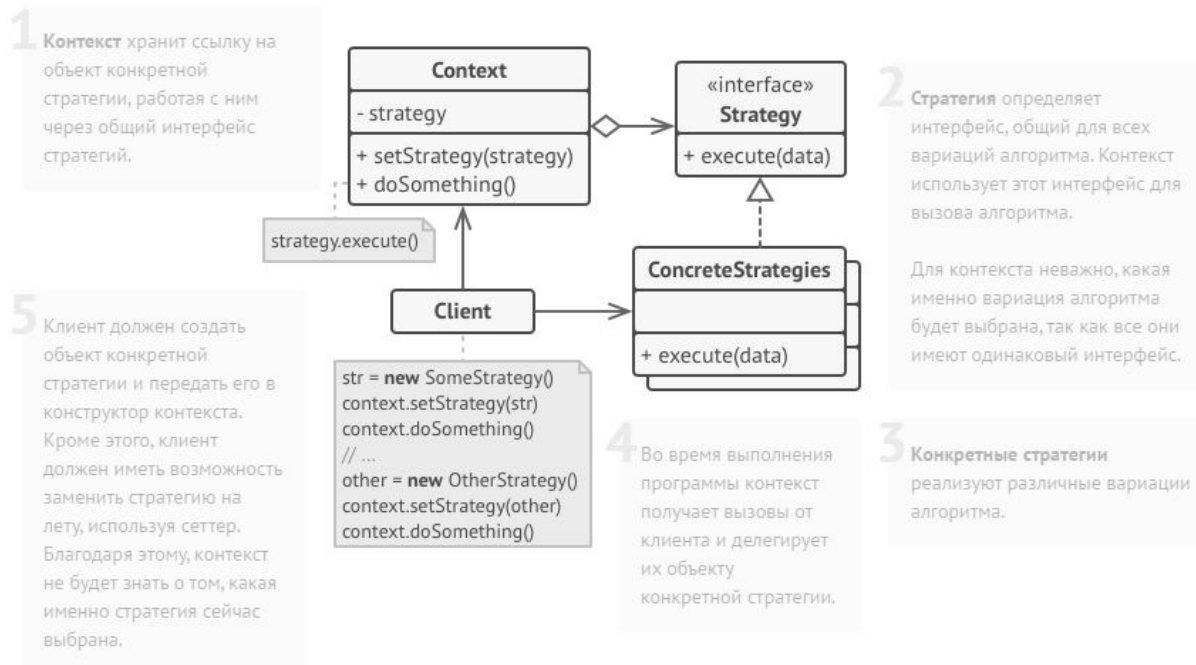


### 33. Паттерн “Стратегия”.

Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

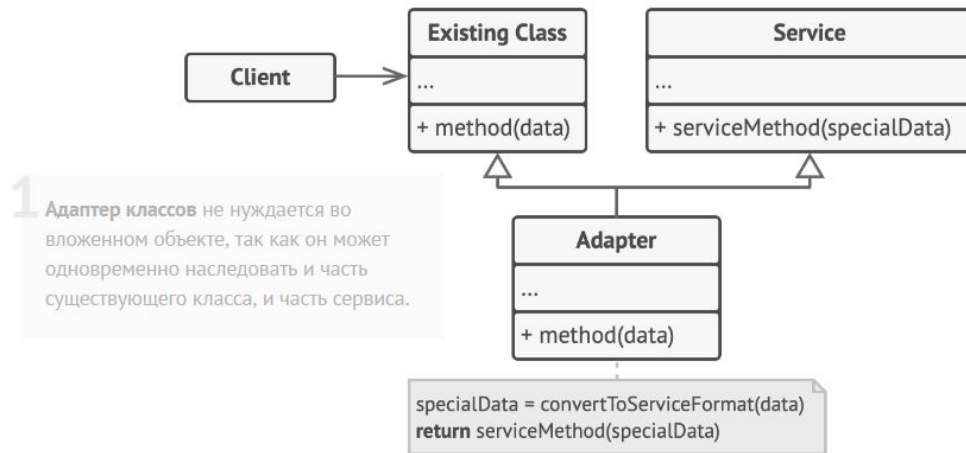


- Назначение — инкапсуляция алгоритма в объект
- Самое важное — спроектировать интерфейсы стратегии и контекста
  - Так, чтобы не менять их для каждой стратегии
- Применяется, если
  - Имеется много родственных классов с разным поведением
  - Нужно иметь несколько вариантов алгоритма
  - В алгоритме есть данные, про которые клиенту знать не надо
  - В коде много условных операторов
- Передача контекста вычислений в стратегию
  - Как параметры метода — уменьшает связность, но некоторые параметры могут быть стратегии не нужны
  - Передавать сам контекст в качестве аргумента — в Context интерфейс для доступа к данным
- Стратегия по умолчанию
  - Или просто поведение по умолчанию, если стратегия не установлена
- Объект-стратегия может быть приспособленцем



## 34. Паттерн “Адаптер”.

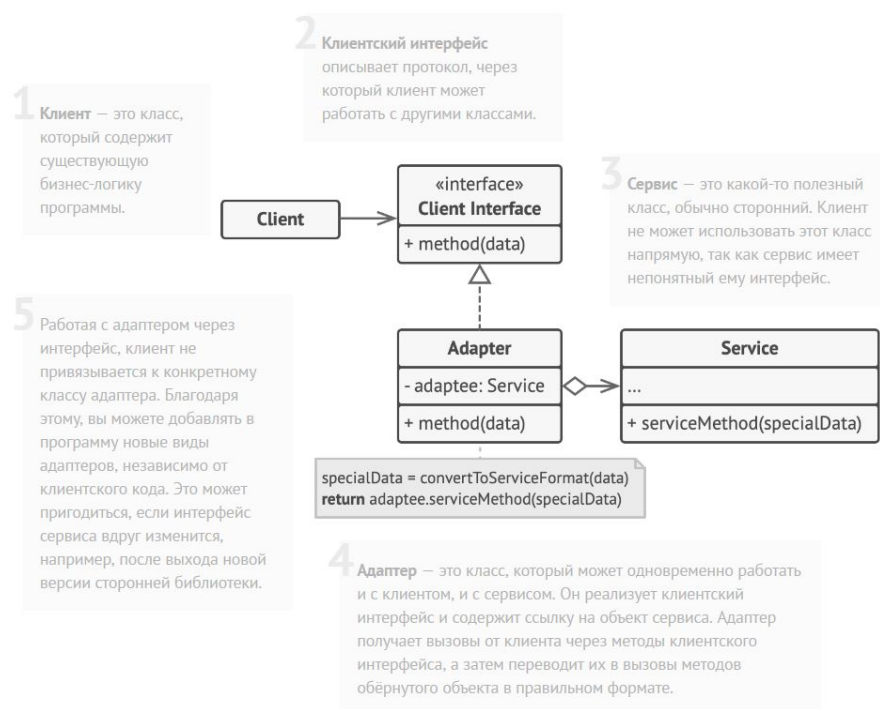
### Адаптер Классов



Использует множественное наследование (от интерфейса и от адаптируемого класса).  
Может реализовывать дополнительную функциональность, который нет в адаптируемом классе.

Можно делать мощные двусторонние адаптеры, но это сложнее.

### Адаптер объектов



Удобно использовать, когда есть много подклассов сервиса, и не хочется делать адаптер классов для каждого из них.

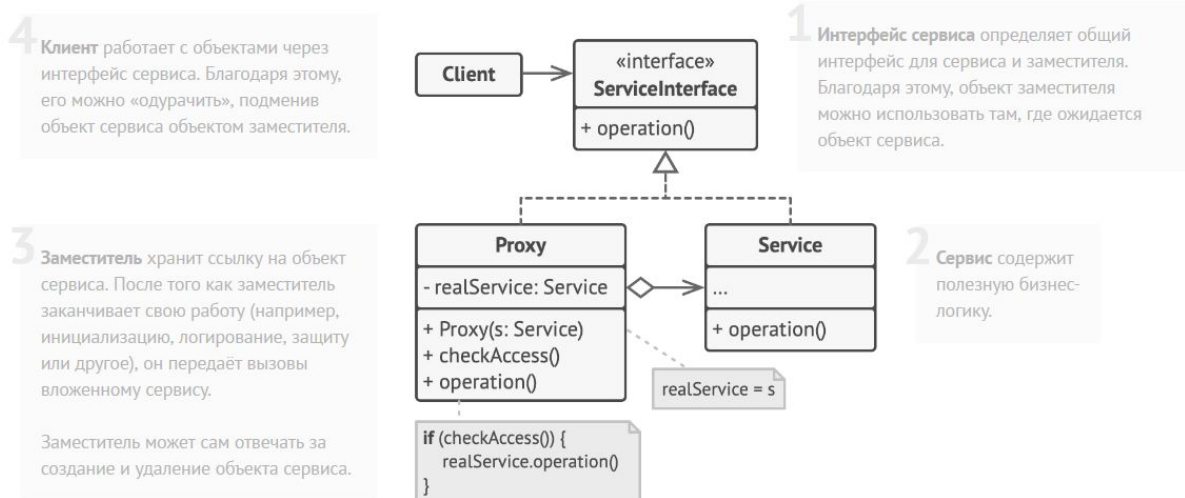
## 35. Паттерн “Прокси”.

### Прокси (Заместитель) (Суррогат)

Вместо требуемого объекта создается объект заместитель, знающий минимально необходимую информацию о требуемом. Объект заместитель может отвечать на простейшие запросы без создания требуемого объекта. Это полезно, если он тяжелый и требуется не сразу.

Пример -- картинки в текстовом редакторе. Нет нужды загружать их, пока пользователь на них еще не посмотрел. Однако информация об их размерах и существовании нужна для рендеринга остальных частей документов.

Так же прокси может использоваться для защиты (проверять доступ перед перенаправлением), логирования, кэширования.

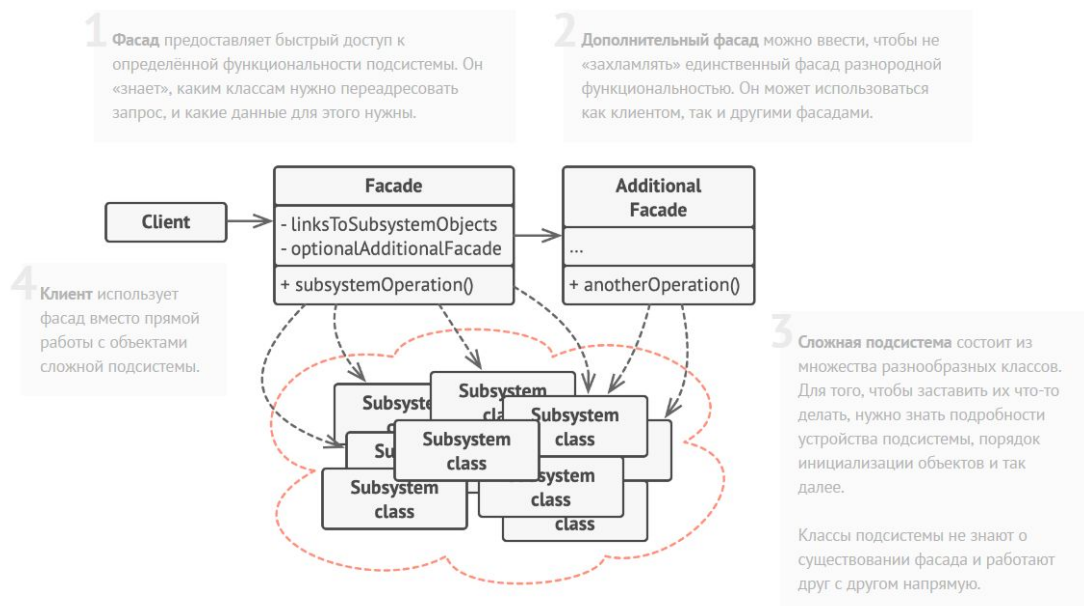


## 36. Паттерн “Фасад”.

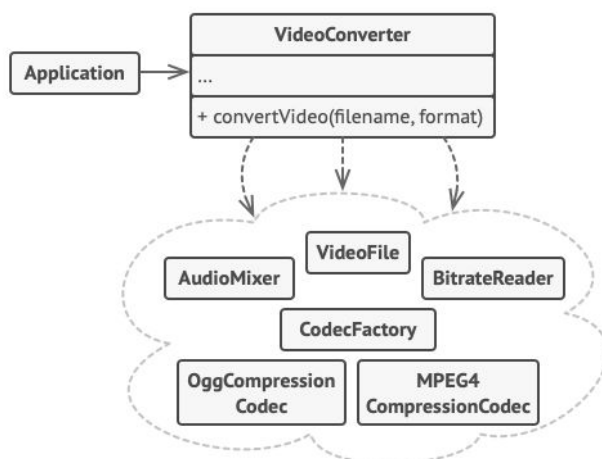
Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

Нужен для упрощения доступа и уменьшения связности.

**NB!** Фасад не обязательно препятствует приложениям напрямую обращаться к классам под-системы, если это необходимо. Таким образом, у вас есть выбор между простотой и общностью.



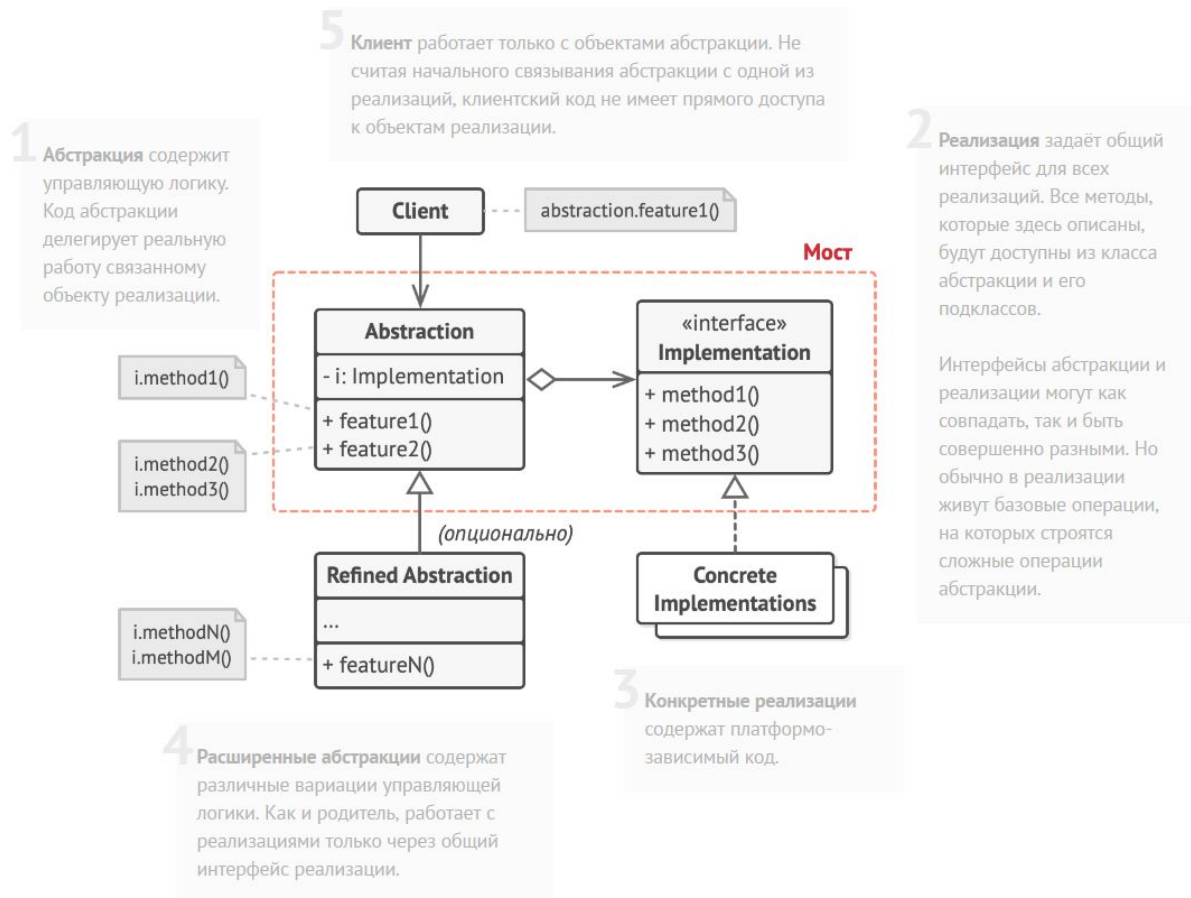
Может быть несколько уровней фасадов для упрощения системы.



Пример изоляции множества зависимостей в одном фасаде.

## 37. Паттерн “Мост”.

**Мост (Handle / Body)** — отделяет абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

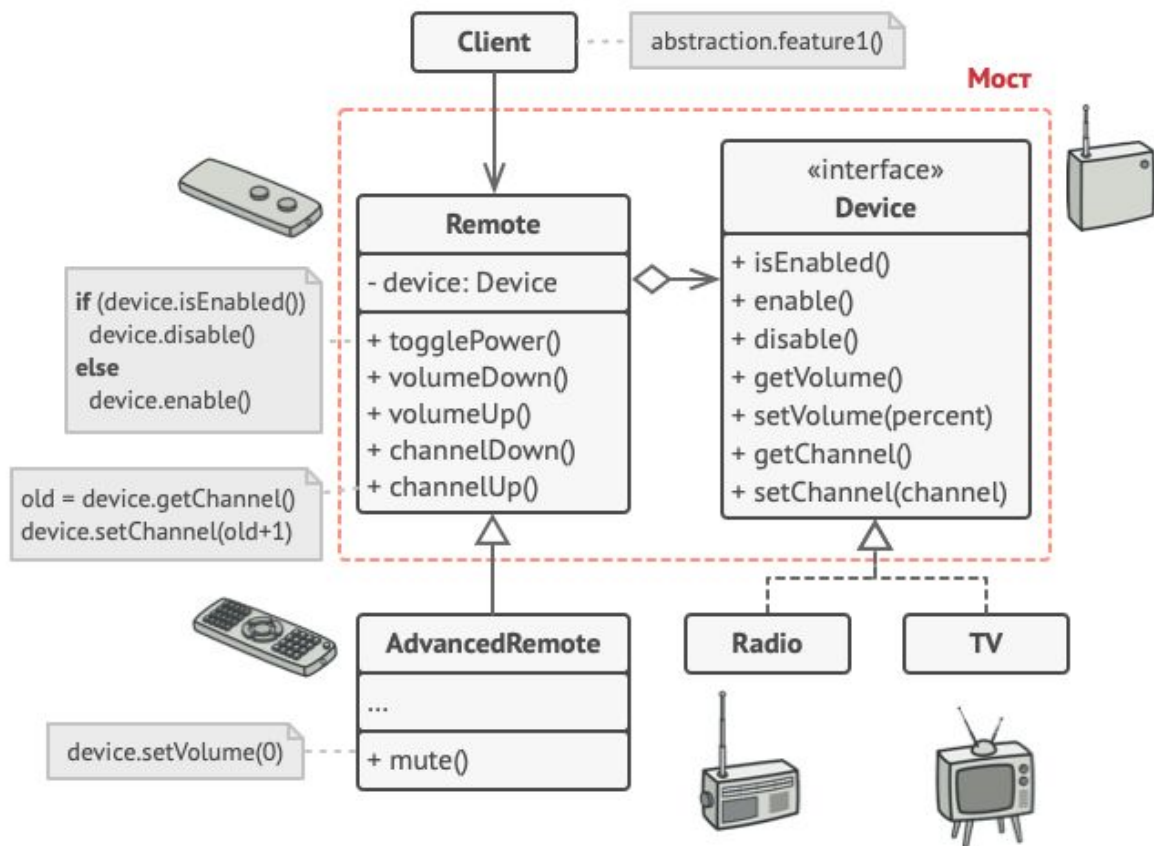


Паттерн применяется, когда систему можно развивать сразу в двух независимых направлениях

- GUI приложения может быть разным для разных типов пользователей
- GUI приложения должно работать на разных платформах

Чтобы не создавать комбинированные подклассы для каждого сценария (типа WinAdminGUI, LinuxUserGUI,...) можно декомпозировать систему на абстракцию (GUI) и реализацию (API, к которому обращается GUI).

Абстракция делегирует работу одной из требуемых реализаций через интерфейс. Поэтому реализации и абстракции можно дополнять независимо.



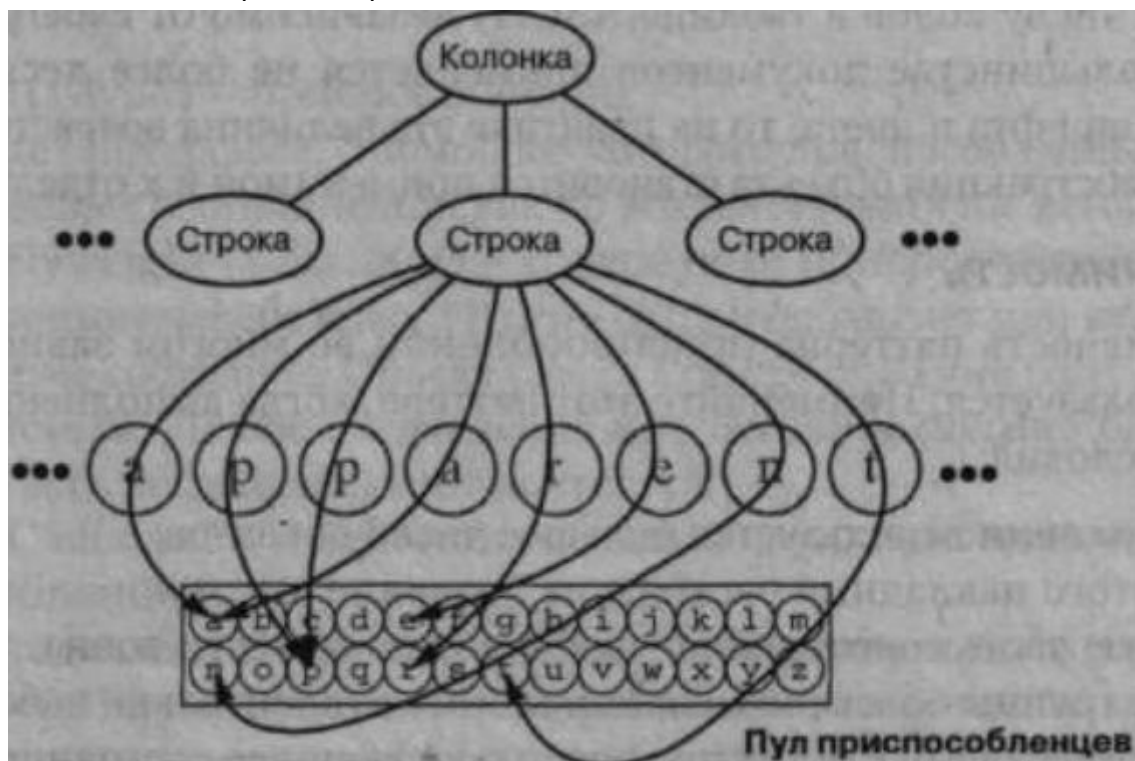
## 38. Паттерн “Приспособленец”.

**Приспособленец (легковес)** — это разделяемый объект, который можно использовать одновременно в нескольких контекстах. В каждом контексте он выглядит как независимый объект, то есть неотличим от экземпляра, который не разделяется.

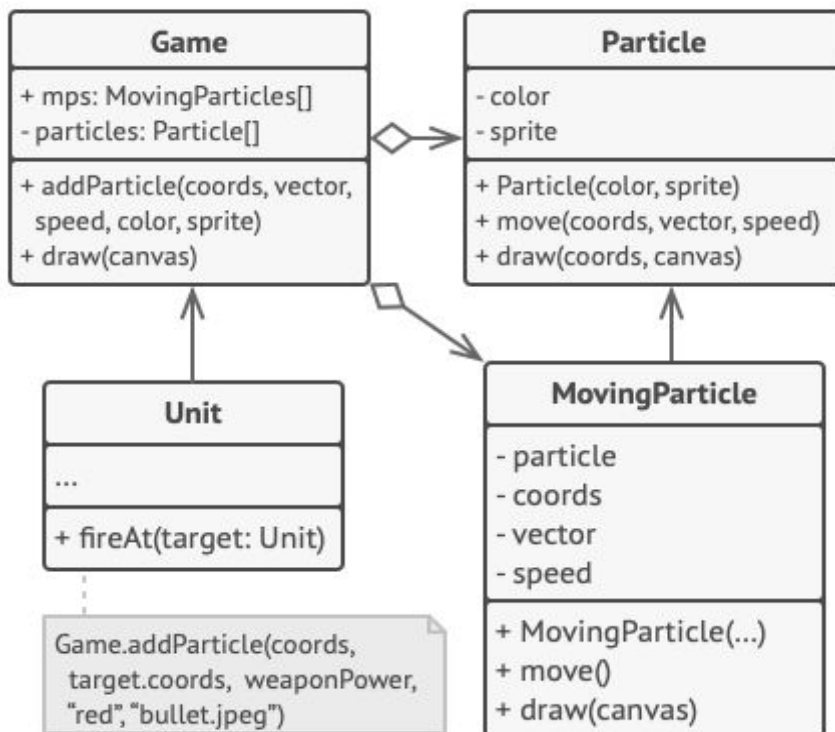
Приспособленцы нужны чтобы сократить затраты памяти:





- Текстовый редактор. Не обязательно хранить код символа для каждой буквы в тексте документа. Можно создать один объект на каждую букву алфавита и ссылаться на него, где требуется.

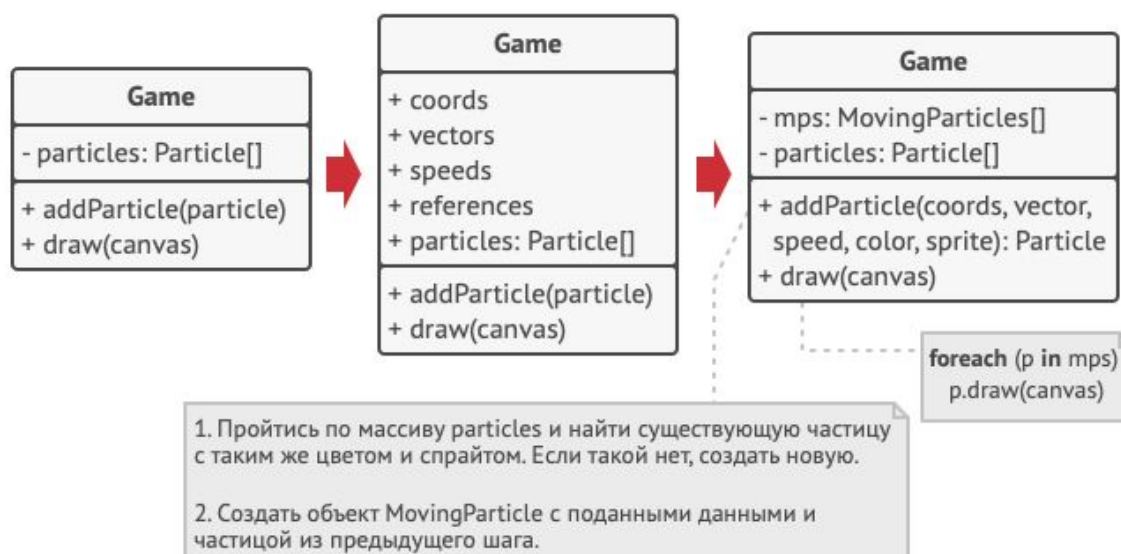
Внутреннее состояние объекта приспособленца должно быть неизменяемым (код символа буквы). Внешнее состояние приспособленца (цвет буквы, размер шрифта,...) может изменяться, но должно храниться вне объекта приспособленца — в объекте контексте. На картинке строка — контекст.







<b>Память</b>	coords: 8B	 × 1
color: 4B	vector: 16B	 × 1.000.000
sprite: 20KB	speed: 4B	
-----	particle: 4B	
 ≈ 21KB	 ≈ 32B	<b>32MB</b>

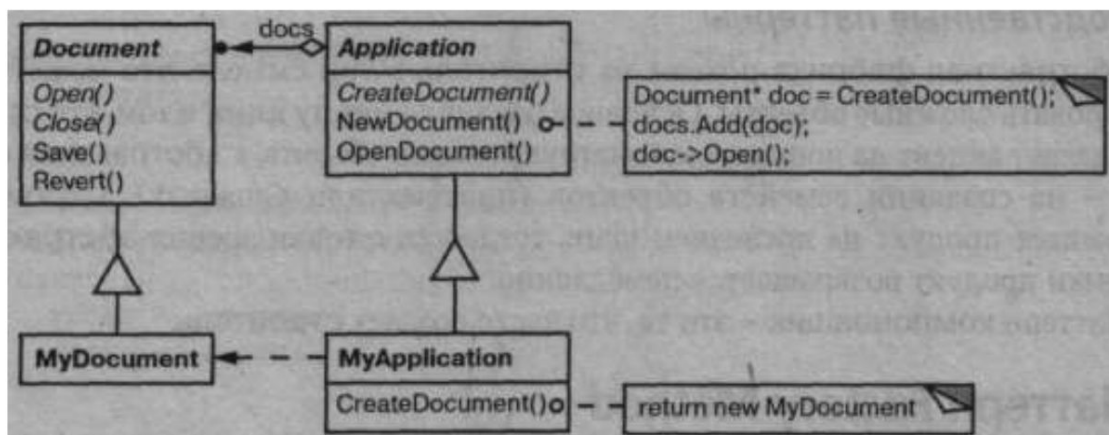
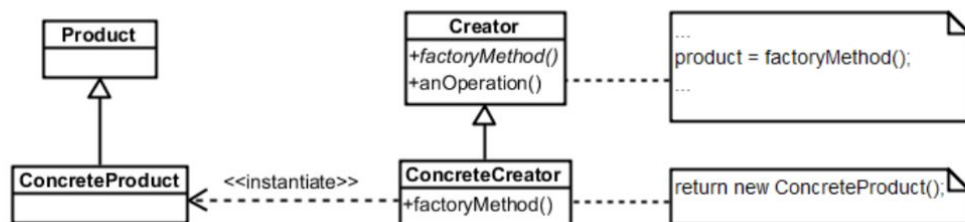


## 39. Паттерн “Спецификация”.

Было

## 40. Паттерн “Фабричный метод”.

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.



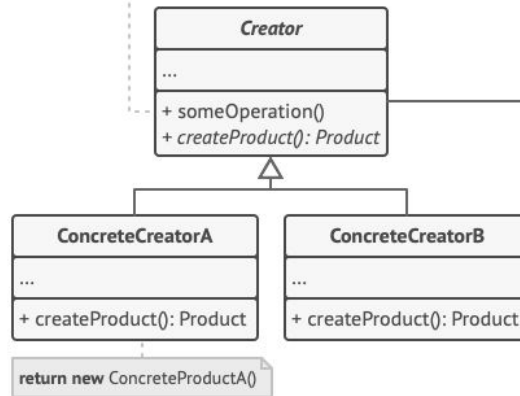
- классу заранее неизвестно, объекты каких классов ему нужно создавать
- объекты, которые создает класс, специфицируются подклассами
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов
- Абстрактный Creator или реализация по умолчанию
  - Второй вариант может быть полезен для расширяемости
- Можно сделать фабричный метод более гибким, позволив ему принимать параметры
- Если язык поддерживает инстанциацию по прототипу (JavaScript, Smalltalk), можно хранить порождаемый объект
- Creator не может вызывать фабричный метод в конструкторе. Иначе, наследующий класс в своем конструкторе тоже **вызовет** фабричный метод родителя (в c++)
  - Можно избежать, если сделать отложенную инициализацию объекта по требованию
- Можно сделать шаблонный Creator (см. следующий билет)

**3** Создатель объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов.

Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Но он может возвращать и некий стандартный продукт.

Несмотря на название, важно понимать, что создание продуктов **не является** единственной функцией создателя. Обычно он содержит и другой полезный код работы с продуктом. Аналогия: большая софтверная компания может иметь центр подготовки программистов, но основная задача компании — создавать программные продукты, а не готовить программистов.

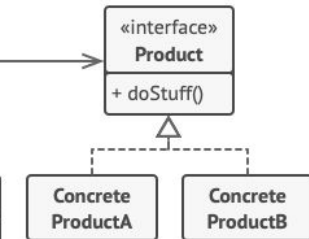
```
Product p = createProduct()
p.doStuff()
```



**4** Конкретные создатели по-своему реализуют фабричный метод, производя те или иные конкретные продукты.

Фабричный метод не обязан всё время создавать новые объекты. Его можно переписать так, чтобы возвращать существующие объекты из какого-то хранилища или кэша.

**1** Продукт определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.



**2** Конкретные продукты содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.

## 41. Паттерн “Шаблонный метод”.

**Шаблонный метод** — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекадывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

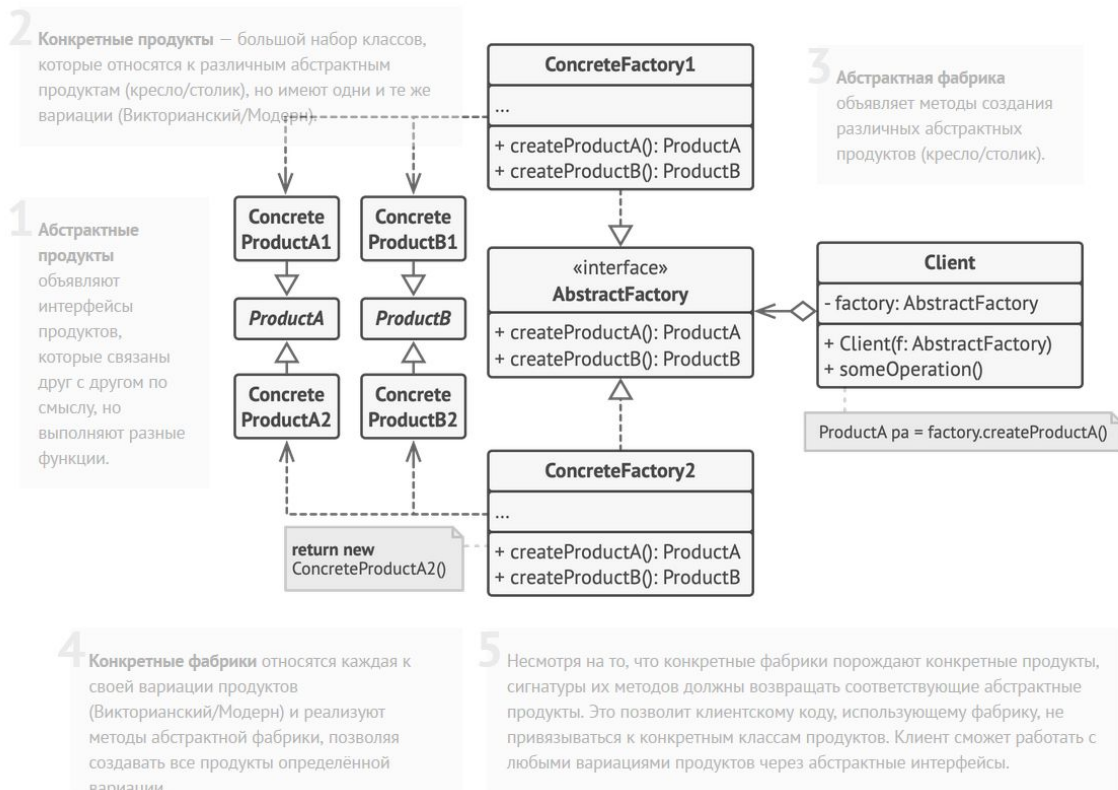


Идея в разбиении алгоритма на последовательность шагов и инкапсуляции его в отдельный класс. Подклассы переопределяют только некоторые шаги алгоритма (например, разные подклассы могут работать с разными типами данных и переопределять этап чтения). Это уменьшает количество кода.

- Жесткая привязка к структуре алгоритма
- Принцип подстановки БЛ может быть нарушен, если переопределять базовые шаги в подклассах

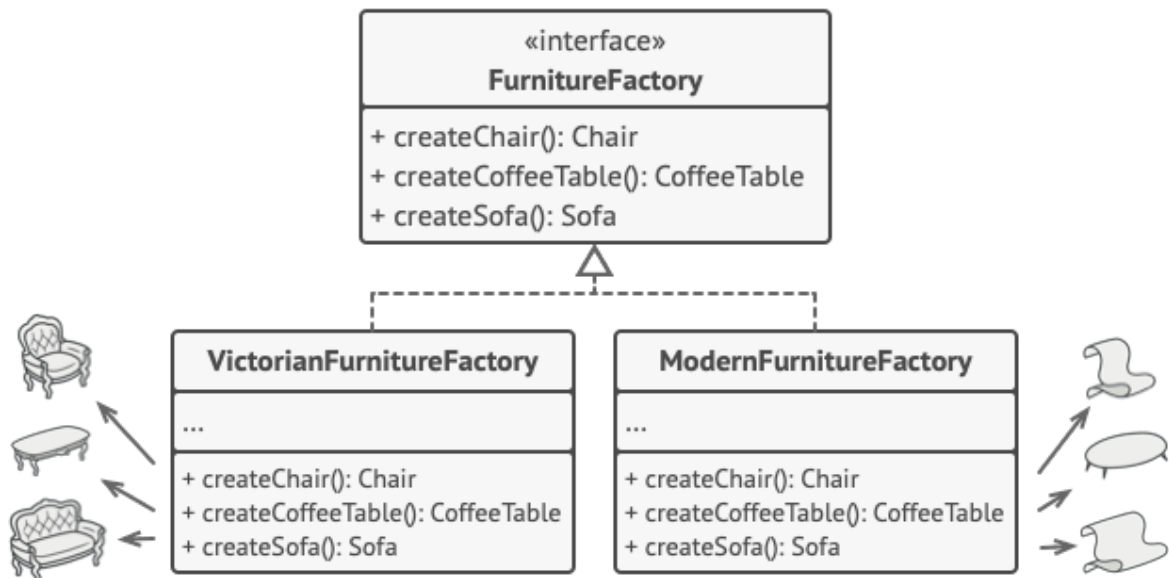
## 42. Паттерн “Абстрактная фабрика”.

**Абстрактная фабрика** — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



- + Изолирует конкретные классы
  - + Упрощает замену семейств продуктов
  - + Гарантирует сочетаемость продуктов
  - Поддержать новый вид продуктов непросто
- 
- Система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты
  - Система должна конфигурироваться одним из семейств составляющих ее объектов
  - Взаимосвязанные объекты должны использоваться вместе
  - Хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию

Пример:

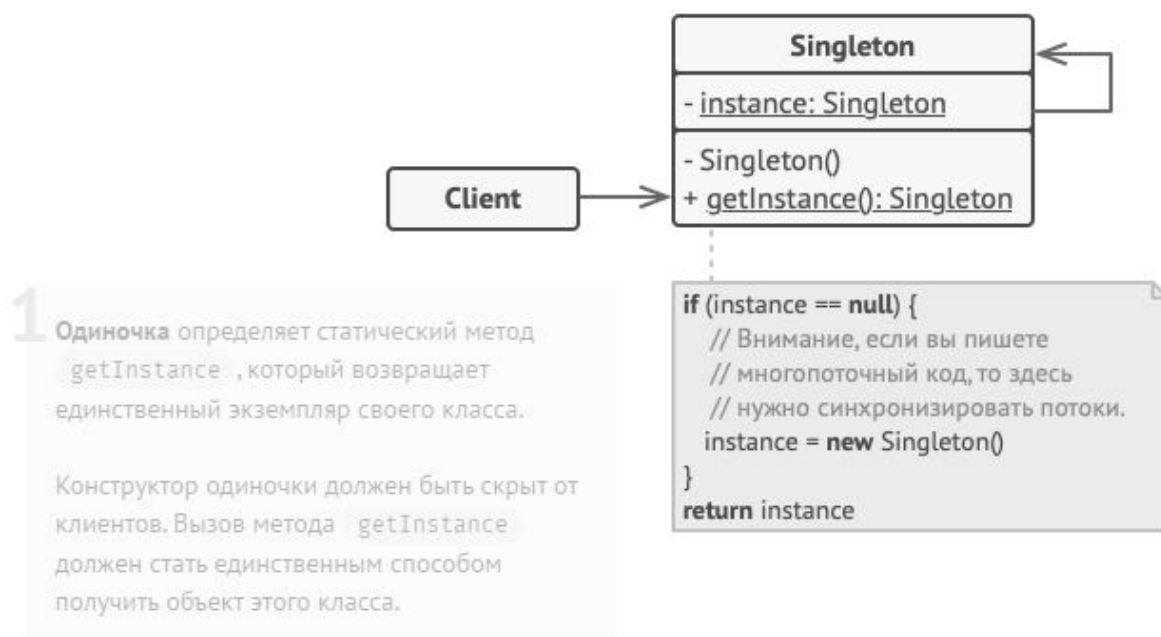


Конкретные фабрики выдают предметы одного стиля.

## 43. Паттерн “Одиночка”.

Паттерн “Одиночка” (Singleton) — позволяет иметь только один экземпляр объекта в системе.

Есть конструктор по умолчанию и публичный статический метод, который всегда отдает один и тот же объект.



Где применять?

- Нужен один экземпляр - например, доступ к базе данных
- Нужна глобальная переменная - синглтон гарантирует, что никто ее не подменит

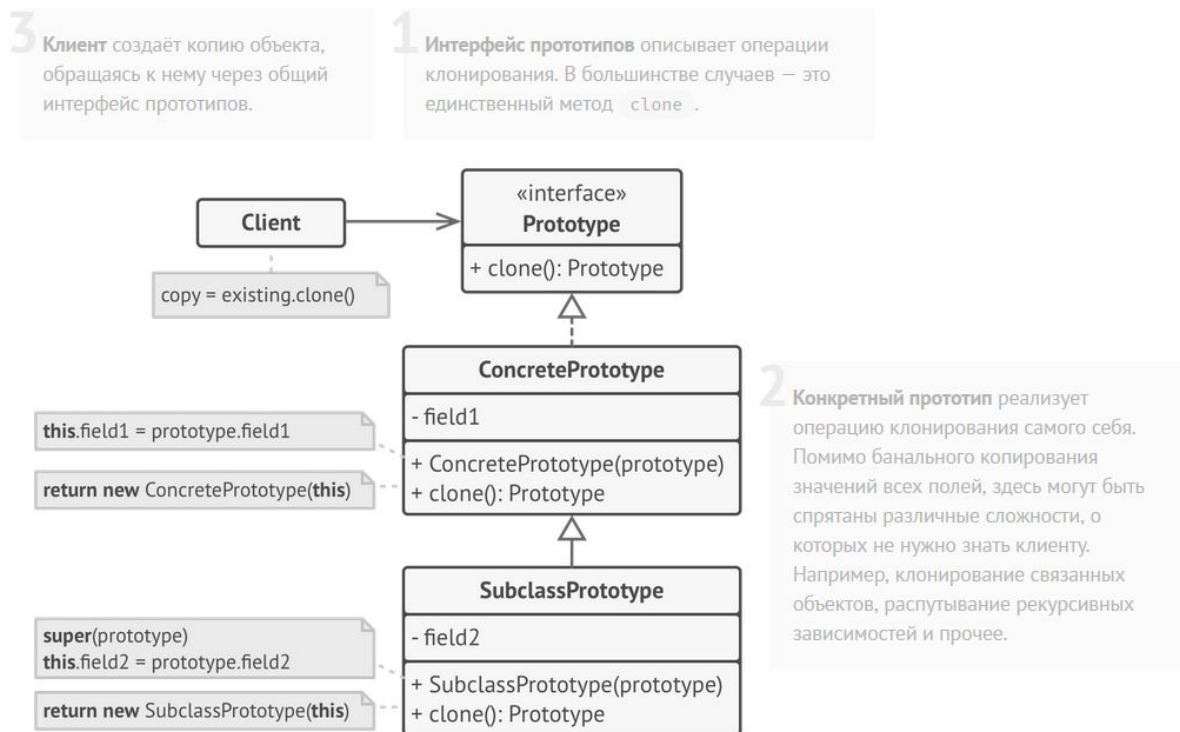
Проблемы:

- [Нарушает](#) принцип единственной ответственности (или [нет](#)) - своя + поддержание инварианта единственности
- Мультипоточность
- Mock - объекты при тестировании
- Повышает связность кода (используется во всей системе)



## 44. Паттерн “Прототип”.

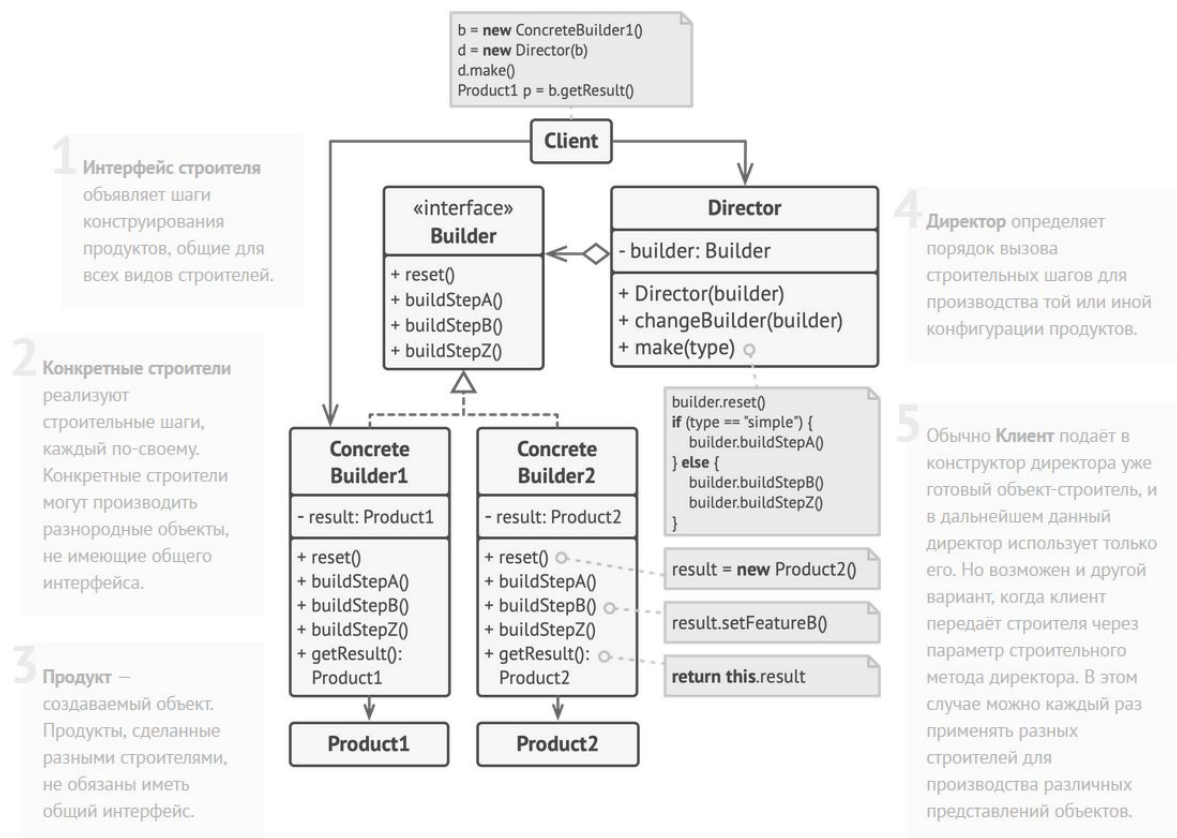
**Прототип** — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.



- Реестр прототипов, обычно ассоциативное хранилище
- Операция Clone
  - Глубокое и мелкое копирование
  - В случае, если могут быть круговые ссылки, будет сложно
  - Сериализовать/десериализовать объект (но помнить про идентичность)
- Инициализация клона
  - Нельзя передавать параметры в `clone`, это раздует интерфейс
  - Вместо этого используют функцию `initialize` на клонированном объекте

## 45. Паттерн “Строитель”.

**Строитель** — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель дает возможность использовать один и тот же код строительства для получения разных представлений объектов.



- Все строители имеют общий интерфейс.
- Если разные строители используют разные методы для создания продукта, можно сделать абстрактный класс, реализующий заглушки методов, вместо интерфейса.
- Объекты, которые производят строители могут иметь разные интерфейсы.
- Можно завести класс Director, чтобы он использовал строителя, а не вызывать методы строителя из клиента.

- + Строитель дает возможность собирать объект по частям
- + Скрывает детали создания объекта и возвращает только полностью готовый объект

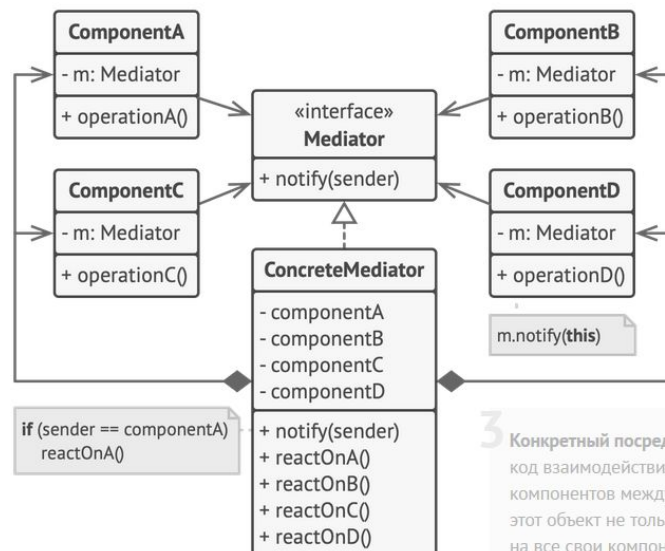
## 46. Паттерн “Посредник”.

**Посредник** — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

**1** Компоненты — это разнородные объекты, содержащие бизнес-логику программы. Каждый компонент хранит ссылку на объект посредника, но работает с ним только через абстрактный интерфейс посредников. Благодаря этому, компоненты можно повторно использовать в другой программе, связав их с посредником другого типа.

**2** Посредник определяет интерфейс для обмена информацией с компонентами. Обычно хватает одного метода, чтобы оповещать посредника о событиях, произошедших в компонентах. В параметрах этого метода можно передавать детали события: ссылку на компонент, в котором оно произошло, и любые другие данные.

**4** Компоненты не должны общаться друг с другом напрямую. Если в компоненте происходит важное событие, он должен оповестить своего посредника, а тот сам решит — касается ли событие других компонентов, и стоит ли их оповещать. При этом компонент-отправитель не знает кто обработает его запрос, а компонент-получатель не знает кто его прислал.



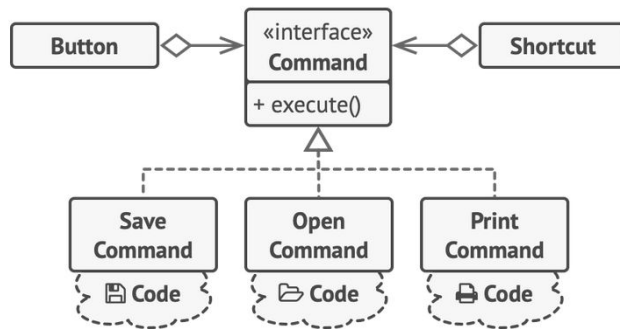
**3** Конкретный посредник содержит код взаимодействия нескольких компонентов между собой. Зачастую этот объект не только хранит ссылки на все свои компоненты, но и сам их создаёт, управляя дальнейшим жизненным циклом.

Компоненты больше не общаются друг с другом, они знают только о посреднике. Посредник получает сигналы от компонент и обрабатывает их с использованием других компонент (может передавать сигнал, может совершать дополнительную работу)

- Уменьшает связность
- Может инкапсулировать в себе слишком много логики и стать нечитаемым

## 47. Паттерн “Команда”.

**Команда** — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.



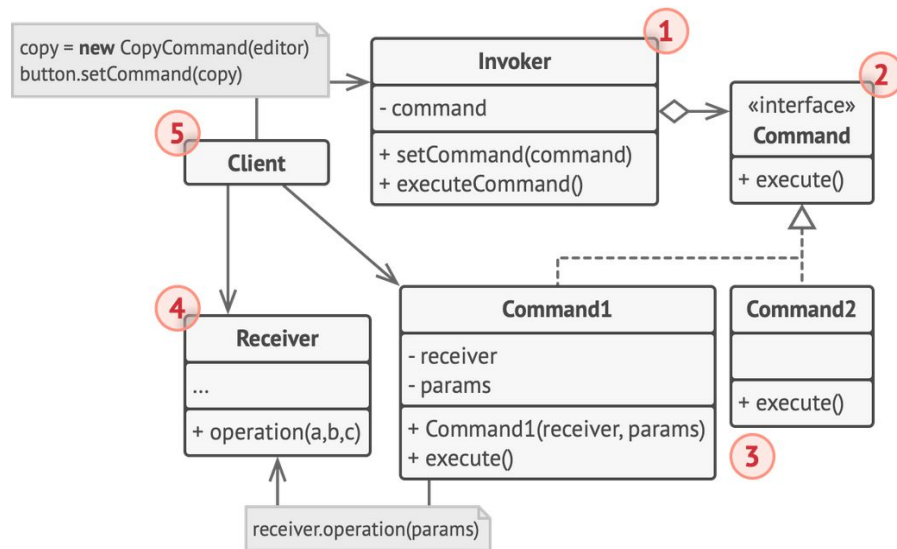
*Классы UI делегируют работу командам.*

- Объекты Button и Shortcut хранят в себе команды. Они не знают ничего о цели и устройстве команды и просто вызывают у нее метод *execute*.
- Аргументы команды хранятся и передаются в поля объекта команды.
- Runtime изменяющийся контекст через команды не передать.
- Сама команда может выполнять какие-то действия самостоятельно, но обычно просто делегирует работу подходящим объектам бизнес логики.
- Команды можно объединять в цепочки или реализовывать составные команды.

Зачем?

- Хотим отделить инициализацию запроса от его исполнения
  - Хотим, чтобы тот, кто “активирует” запрос, не знал, как он исполняется
  - При этом хотим, чтобы тот, кто знает, когда исполнится запрос, не знал, когда он будет активирован
- Примеры
  - Команды меню приложения
  - Палитры инструментов
- “Просто вызвать действие” не получится, вызов функции жёстко свяжет инициатора и исполнителя
- Параметризовать объекты выполняемым действием
- Определять, ставить в очередь и выполнять запросы в разное время
- Поддерживать отмену операций
- Структурировать систему на основе высокоуровневых операций, построенных из примитивных

- Поддержать протоколирование (логирование) изменений



- Насколько “умной” должна быть команда
- Отмена и повторение операций — тоже от хранения всего состояния в команде до “вычислимого” отката
  - Undo-стек и Redo-стек
  - Может потребоваться копировать команды
  - “Искусственные” команды
  - Композитные команды
  - Паттерн “Хранитель” для избежания ошибок восстановления

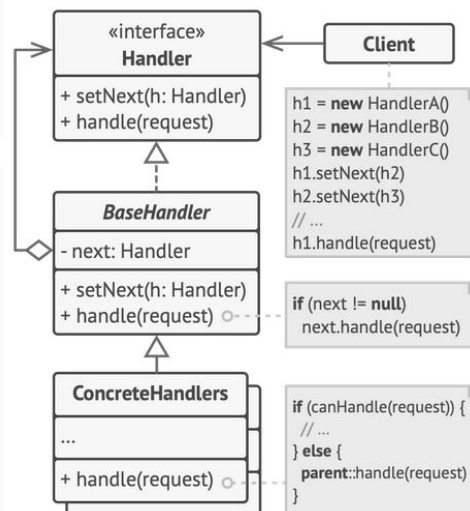
## 48. Паттерн “Цепочка ответственности”.

**Цепочка обязанностей** — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

**1** Обработчик определяет общий для всех конкретных обработчиков интерфейс. Обычно достаточно описать единственный метод обработки запросов, но иногда здесь может быть объявлен и метод выставления следующего обработчика.

**2** Базовый обработчик — опциональный класс, который позволяет избавиться от дублирования одного и того же кода во всех конкретных обработчиках.

Обычно этот класс имеет поле для хранения ссылки на следующий обработчик в цепочке. Клиент связывает обработчики в цепь, подавая ссылку на следующий обработчик через конструктор или сеттер поля. Также здесь можно реализовать базовый метод обработки, который бы просто перенаправлял запрос следующему обработчику, проверив его наличие.



**4** Клиент может либо сформировать цепочку обработчиков единожды, либо перестраивать её динамически, в зависимости от логики программы. Клиент может отправлять запросы любому из объектов цепочки, не обязательно первому из них.

**3** Конкретные обработчики содержат код обработки запросов. При получении запроса каждый обработчик решает, может ли он обработать запрос, а также стоит ли передать его следующему объекту.

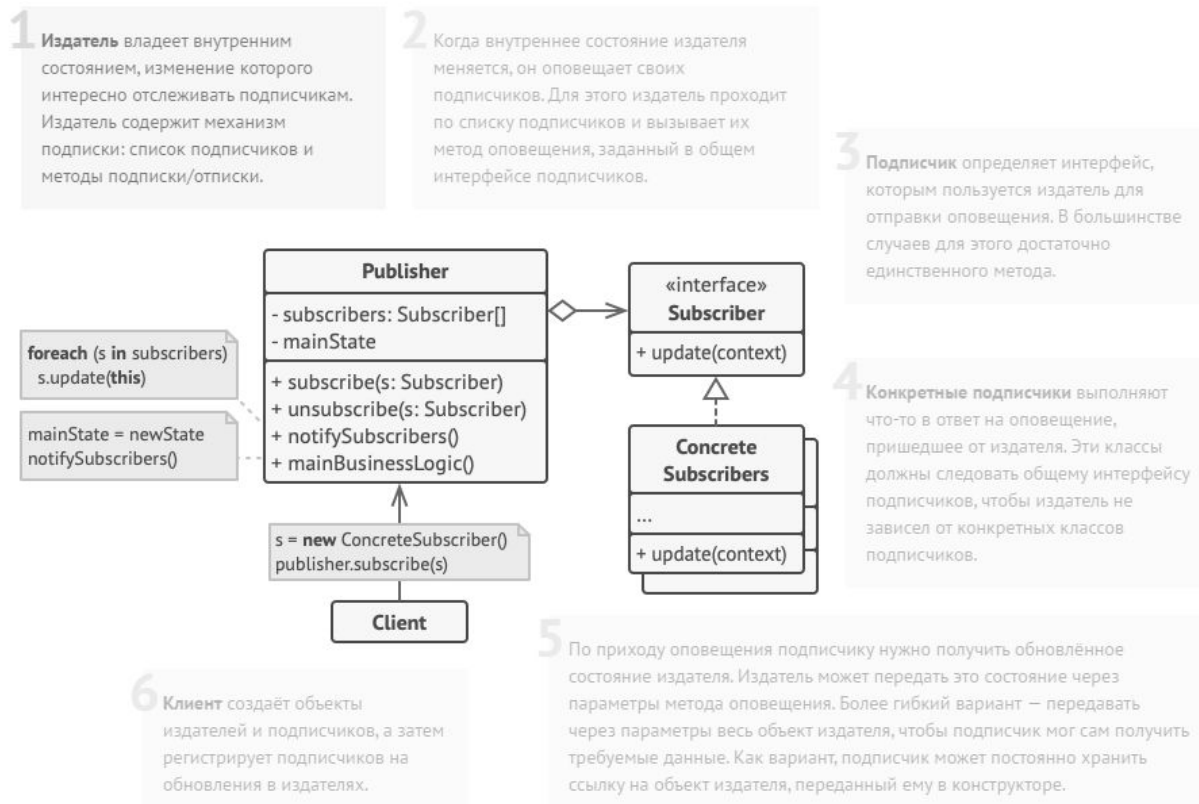
В большинстве случаев обработчики могут работать сами по себе и быть неизменяемыми, получив все нужные детали через параметры конструктора.

- Когда программа должна обрабатывать разнообразные запросы несколькими способами, но заранее неизвестно, какие конкретно запросы будут приходить и какие обработчики для них понадобятся.
- Когда важно, чтобы обработчики выполнялись один за другим в строгом порядке.
- Когда набор объектов, способных обработать запрос, должен задаваться динамически.

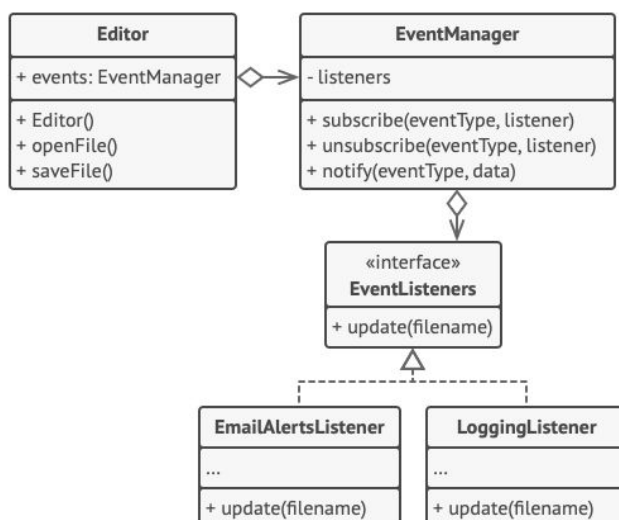
Запрос может никто не обработать, и никто об этом не узнает.

## 49. Паттерн “Наблюдатель”.

*Издатели* содержат интересную *Подписчикам* информацию. *Издатель* предоставляет *Подписчикам* методы добавления себя в списки рассылки. Когда *Издатель* получает информацию, он рассылает ее заинтересованным *Подписчикам*.



Реализует принцип open/closed. Но подписчики извещаются в случайном порядке.

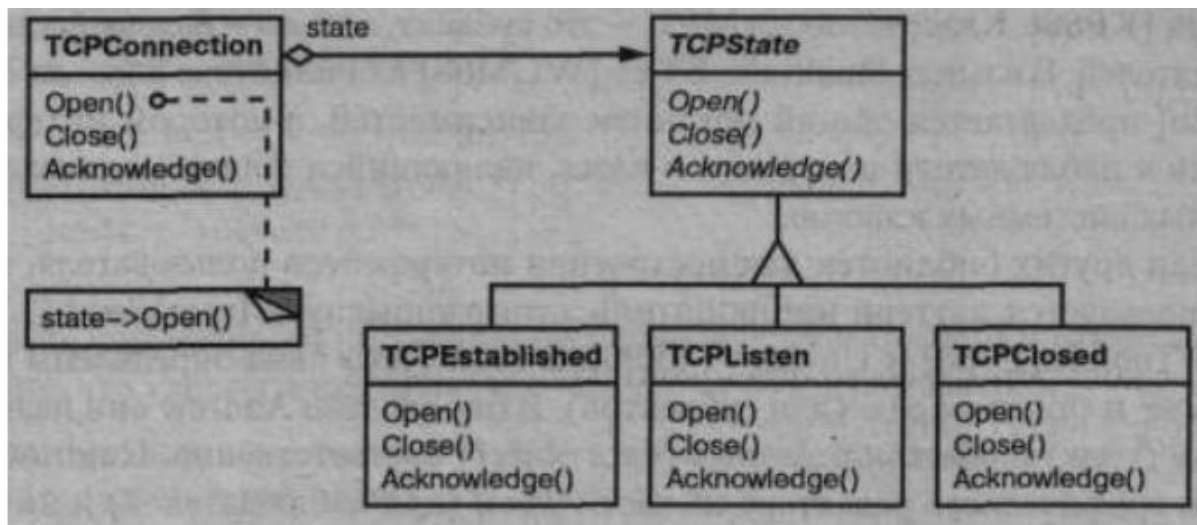


Пример оповещения объектов о событиях в других объектах.



## 50. Паттерн “Состояние”.

**Состояние** — это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создается впечатление, что изменился класс объекта.



Используйте паттерн состояние в следующих случаях:

- когда поведение объекта зависит от его состояния и должно изменяться во время выполнения;
- когда в коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви зависит от состояния. Паттерн “Состояние” предлагает поместить каждую ветвь в отдельный класс. Это позволяет трактовать состояние объекта как самостоятельный объект, который может изменяться независимо от других.

Преимущества:

- + Локализует зависящее от состояния поведение и делит его на части, соответствующие состояниям.
- + Делает явными переходы между состояниями.
- + Объекты состояния можно разделять.

Вопросы реализации:

- Переходы между состояниями
  - Context сам определяет, когда менять состояние и соответствующий класс
  - State получает специальный интерфейс, чтобы самостоятельно менять класс состояния в Context
- Таблица переходов — вместо классов, ищем нужное состояние в таблице
  - Поиск часто менее эффективен, чем вызов функций
  - Табличный формат тяжелее понимать
  - Трудно добавить действия по переходу



- Создание и уничтожение состояний
  - Создать раз и навсегда
  - Создавать и удалять при переходах

## 51. Паттерн “Посетитель”.

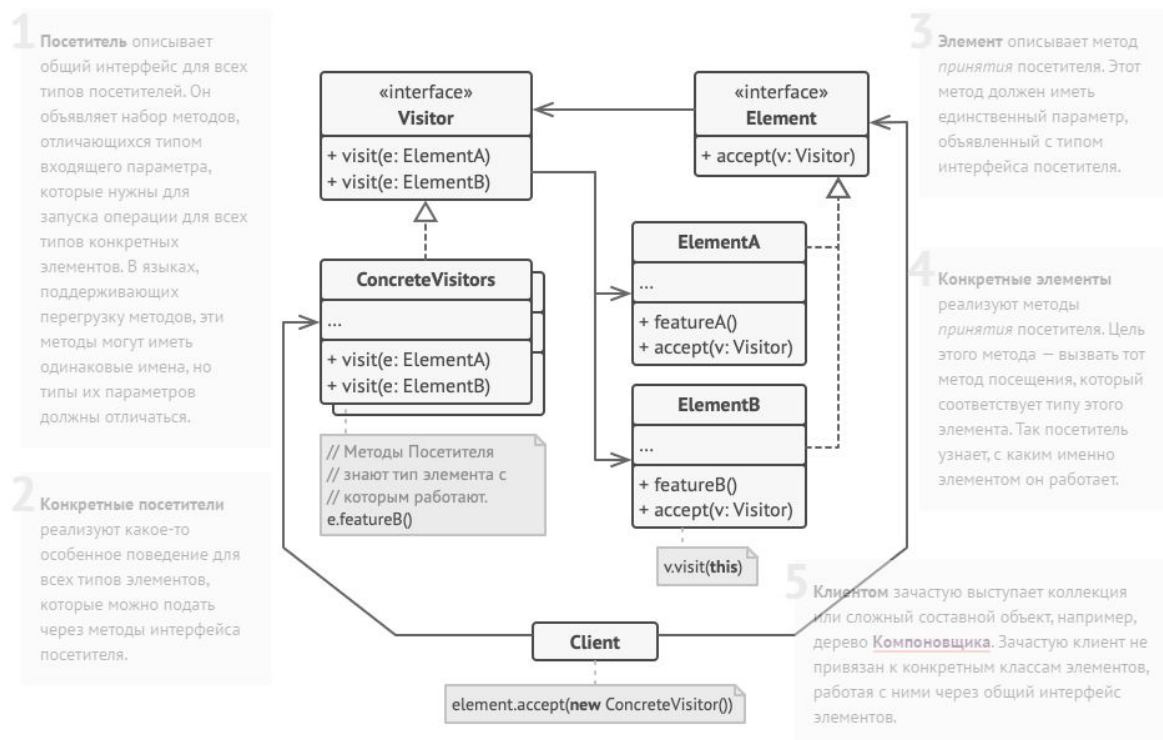
**Посетитель** — поведенческий шаблон, позволяющий отделить алгоритмы от объектов, с которыми они работают.

**Мотивация** — необходимо совершать какие-то новые операции с объектами уже существующих классов.

Добавлять новое поведение в эти классы нельзя, потому что:

- нежелательно модифицировать код уже использующихся в системе объектов (и потенциально ломать работающую программу);
- новые операции могут нарушать принцип единственности ответственности и вообще быть не к месту;

Решение — используя трюк [Double dispatch](#), создаем новый класс, который будет “посещать” нужные объекты и совершать с ними необходимые действия.



Преимущества:

- + Соблюдение принципа Open/Closed (новое поведение добавляется путем расширения, а не модификации)
- + Соблюдение принципа Single Responsibility (через наследование можем реализовать другие разновидности того же поведения)
- + Можно накапливать нужную информацию при обходе объектов (например, при обходе дерева)

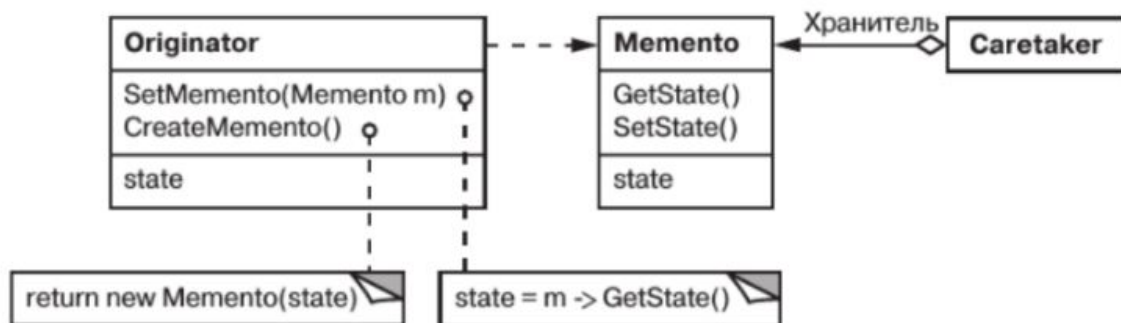
Недостатки:

- Необходимо изменять код всех посетителей, если какой-то вид посещаемых объектов исчез из системы или был в нее добавлен
- У посетителей может не быть доступа к приватным полям/методам обрабатываемых объектов

## 52. Паттерн “Хранитель”.

**Хранитель** — поведенческий паттерн, который, не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позднее можно было восстановить в нем объект.

**Мотивация** — хотим реализовать механизм контрольных точек/откатов, но не хотим раскрывать внутреннее устройство внешним сущностям.



Устройство:

- **Memento (хранитель):**
  - сохраняет внутреннее состояние объекта **Originator**;
  - запрещает доступ всем другим объектам, кроме хозяина. По существу, имеет два интерфейса: широкий — для хозяина, узкий — для посыльного (**Caretaker**);
- **Originator (хозяин):**
  - создает хранителя со снимком текущего состояния;
  - использует хранителя для восстановления внутреннего состояния;
- **Caretaker (посыльный):**
  - отвечает за сохранение хранителя;
  - не производит никаких операций над хранителем и не исследует его внутреннее содержимое.

Преимущества:

- + Можем создавать снимки состояния объекта без нарушения инкапсуляции
- + Хранение истории объекта выносится в хранителя, поэтому код хозяина может стать проще

Недостатки:

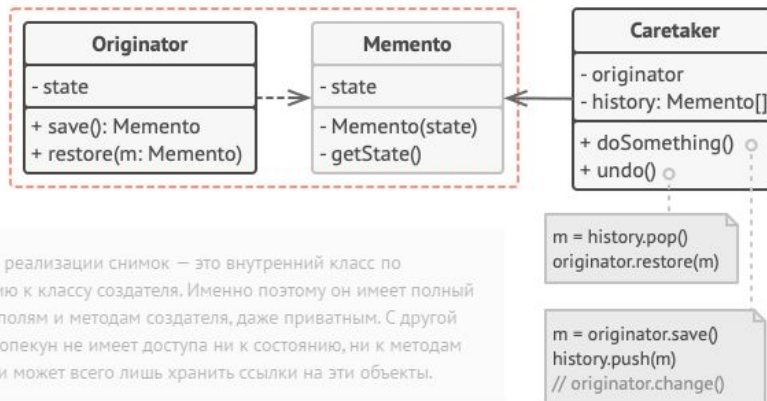
- Большие затраты памяти, если снимки создаются часто
- Посыльные должны следить за жизненным циклом хозяина, чтобы уничтожать невалидные снимки

**1 Создатель** может производить снимки своего состояния, а также воспроизводить прошлое состояние, если подать в него готовый снимок.

**2 Снимок** — это простой объект данных, содержащий состояние создателя. Надёжнее всего сделать объекты снимков неизменяемыми, передавая в них состояние только через конструктор.

**3 Опекун** должен знать, когда делать снимок создателя и когда его нужно восстанавливать.

Опекун может хранить историю прошлых состояний создателя в виде стека из снимков. Когда понадобится отменить выполненную операцию, он возьмёт «верхний» снимок из стека и передаст его создателю для восстановления.



**4** В данной реализации снимок — это внутренний класс по отношению к классу создателя. Именно поэтому он имеет полный доступ к полям и методам создателя, даже приватным. С другой стороны, опекун не имеет доступа ни к состоянию, ни к методам снимков и может всего лишь хранить ссылки на эти объекты.

## 53. Архитектура распределенных систем: понятие распределенной системы, типичные архитектурные стили.

В распределенных системах

- Компоненты приложения находятся в компьютерной сети
- Взаимодействуют через обмен сообщениями
- Основное назначение — работа с общими ресурсами

Особенности:

- Параллельная работа
- Независимые отказы
- Отсутствие единого времени

Заблуждения:

- Сеть надежна
- Задержка = 0
- Пропускная способность бесконечна
- Сеть безопасна
- Топология сети неизменна
- Администрирование сети централизованно
- Передача данных “бесплатна”
- Сеть однородна

Типичные архитектурные стили:

- Уровневая архитектура
  - разделяются функции представления, обработки и хранения данных
  - ОС
  - Коммуникационная инфраструктура (Middleware)
  - Приложения и сервисы
- Клиент-сервер
  - Тонкий клиент
  - Бизнес-логика и данные — на сервере
- Трёхзвенная и N-уровневая (?) архитектуры
  - Бизнес-логику и работу с данными часто разделяют

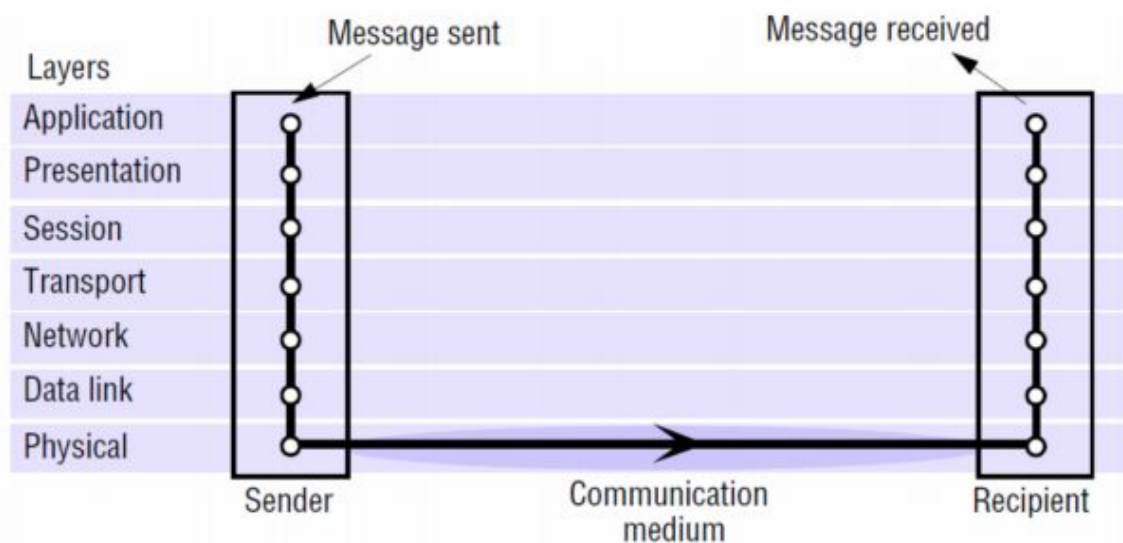
Уровень - механизм физического структурирования инфраструктуры системы. Слой - механизм логического структурирования компонентов, из которых состоит программное решение.

## 54. Межпроцессное сетевое взаимодействие, модель OSI, стек протоколов TCP/IP, сокеты, протоколы “запрос-ответ”.

### Межпроцессное взаимодействие

- Удаленные вызовы
  - Протоколы вида “запрос-ответ”
    - Удаленные вызовы процедур (remote procedure calls, RPC)
    - Удаленные вызовы методов (remote method invocation, RMI)
- Неявное взаимодействие
  - Групповое взаимодействие
    - Модель “издатель-подписчик”
    - Очереди сообщений
    - Распределенная общая память

**Сетевая модель OSI (The Open Systems Interconnection model)** — сетевая модель стека (магазина) сетевых протоколов OSI/ISO. Посредством данной модели различные сетевые устройства могут взаимодействовать друг с другом. Модель определяет различные уровни взаимодействия систем. Каждый уровень выполняет определенные функции при таком взаимодействии.

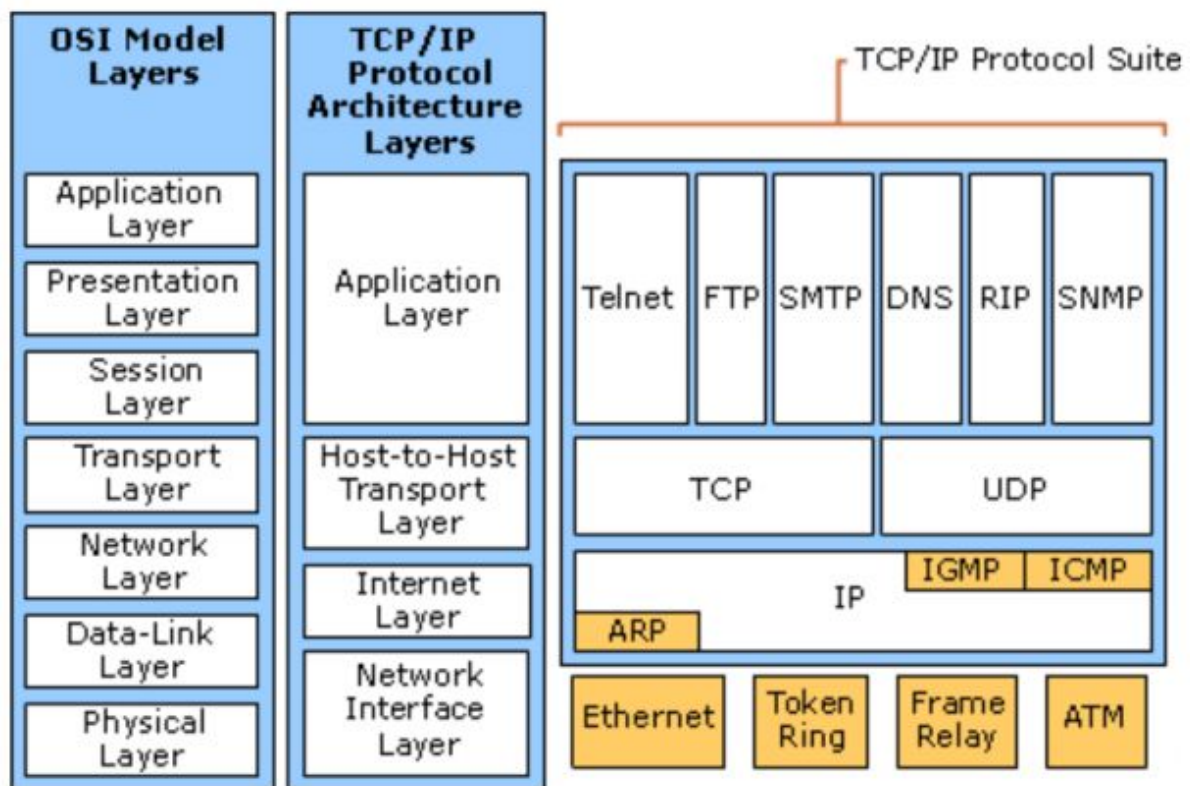


### Уровни:

1. **Application Layer** — верхний уровень модели, обеспечивающий взаимодействие пользовательских приложений с сетью.
2. **Presentation Layer** — обеспечивает преобразование протоколов и кодирование/декодирование данных.
3. **Session Layer** — обеспечивает поддержание сеанса связи, позволяя приложениям взаимодействовать между собой длительное время.
4. **Transport Layer** — предназначен для обеспечения надежной передачи данных от отправителя к получателю.

5. **Network Layer** — предназначен для определения пути передачи данных. Отвечает за трансляцию логических адресов и имён в физические, определение кратчайших маршрутов, коммутацию и маршрутизацию, отслеживание неполадок и «заторов» в сети.
6. **Data Link Layer** — предназначен для обеспечения взаимодействия сетей на физическом уровне и контроля ошибок, которые могут возникнуть. Полученные с физического уровня данные, представленные в битах, он упаковывает в кадры, проверяет их на целостность и, если нужно, исправляет ошибки (формирует повторный запрос поврежденного кадра) и отправляет на сетевой уровень.
7. **Physical Layer** — определяет метод передачи данных, представленных в двоичном виде, от одного устройства (компьютера) к другому.

**TCP/IP** — сетевая модель передачи данных, представленных в цифровом виде. Модель описывает способ передачи данных от источника информации к получателю. В модели предполагается прохождение информации через четыре уровня, каждый из которых описывается правилом (протоколом передачи). Наборы правил, решающих задачу по передаче данных, составляют стек протоколов передачи данных, на которых базируется Интернет.



Уровни:

1. **Application layer** (слои Application, Presentation и Session из OSI) — здесь работает большинство сетевых приложений. Протоколы: HTTP, RTSP, FTP, DNS.



2. **Transport Layer** — здесь решается проблема негарантированной доставки сообщений и правильной последовательности прихода данных. Протоколы: TCP, UDP, SCTP, DCCP.
3. **Internet Layer** — для передачи данных из одной сети в другую. На этом уровне работают маршрутизаторы, которые перенаправляют пакеты в нужную сеть путём расчёта адреса сети по маске сети. Протоколы: IP, ICMP, IGMP, ARP (он между канальным и сетевым).
4. **Network Access Layer** — описывает способ кодирования данных для передачи пакета данных на физическом уровне. Протоколы: Ethernet, IEEE 802.11 WLAN, Token Ring и др.

**Сокет** — название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут быть реализованы как на одном хосте, так и на различных хостах, связанных между собой сетью.

**Сокет** — абстрактный объект, представляющий собой конечную точку соединения.

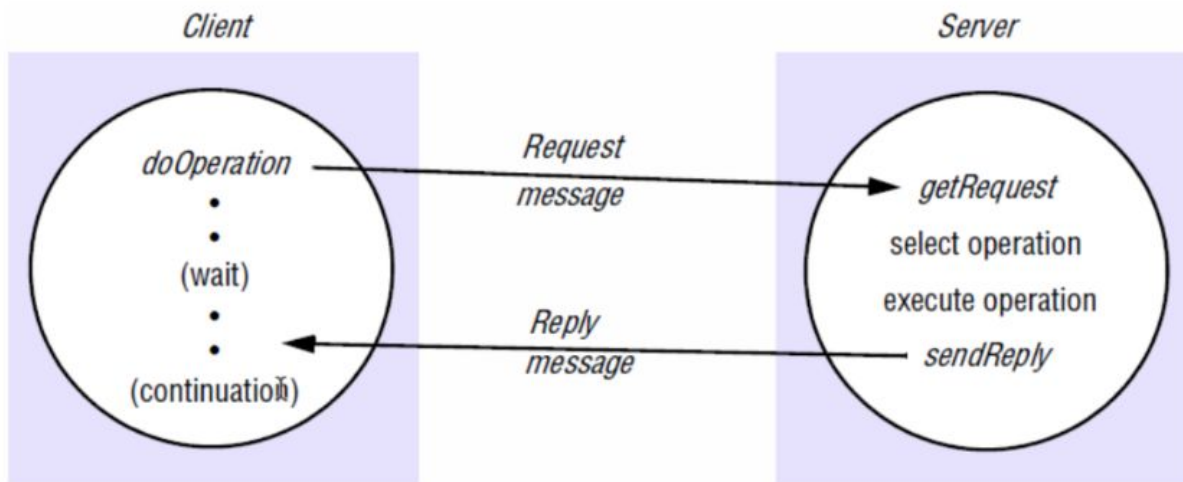
Интерфейс сокетов впервые появился в BSD Unix. Программный интерфейс сокетов описан в стандарте POSIX.1 и в той или иной мере поддерживается всеми современными операционными системами.

В TCP/IP **сокет** определяется парой из IP и порта.

*\*продолжение билета далее\**

### Протоколы “запрос-ответ”

- Синхронные вызовы
- Обмен через request message и reply message



### Запрос-ответ поверх UDP:

- + Уведомления не нужны
- + Установление соединения не нужно
- + Управление потоком не имеет смысла (порядок все равно не определен)
- Неопределённый порядок пакетов
- Потери пакетов
  - Таймаут + повторный запрос нужно реализовывать на уровне бизнес-логики
  - Защита от повторного выполнения операции (хранение “истории”) (пакет может прийти дважды)
  - Новый запрос как подтверждение получения прошлого

### Запрос-ответ поверх TCP:

- + Использование потоков вместо набора пакетов
  - + Удобная отправка больших объёмов данных
  - + Один поток на всё взаимодействие
- + Интеграция с потоками ОО-языков (Объектно-Ориентированных)
- + Надёжность доставки
  - + Отсутствие необходимости проверок на уровне бизнес-логики
  - + Уведомления в пакетах с ответом
  - + Упрощение реализации
- Тяжеловесность коммуникации

## 55. Удаленные вызовы процедур (RPC), удаленные вызовы методов (RMI). Protobuf, gRPC.

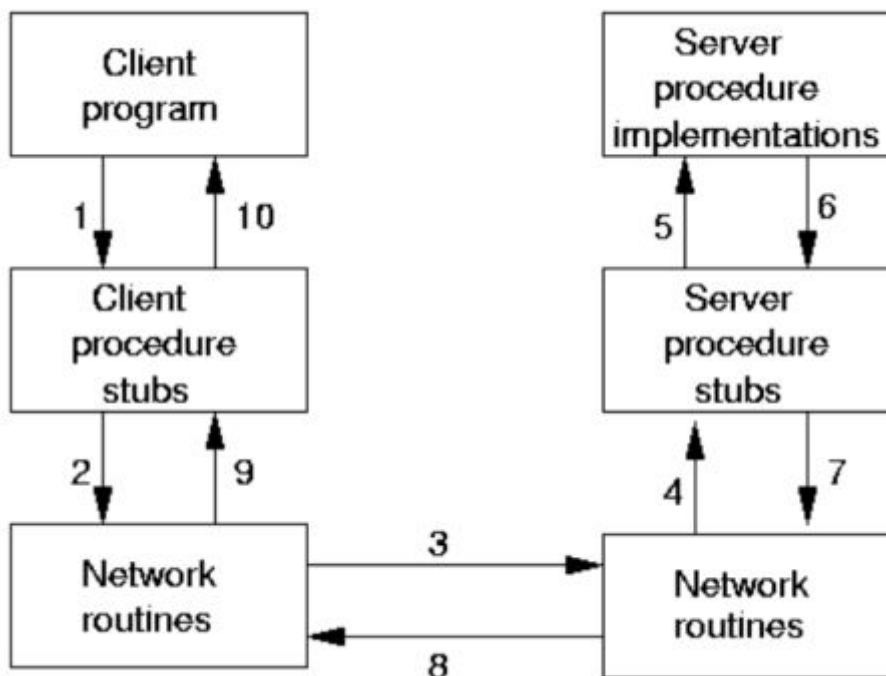
**Стаб** - фрагмент программного кода, используемый для конвертации параметров, передаваемых при помощи RPC (*remote procedure call*)

**RPC** задизайнены так, чтобы быть максимально похожими на обычные вызовы. Поэтому необходимо удовлетворять двум свойствам:

- **Access Transparency** — интерфейсы удаленных вызовов идентичны локальным (нельзя отличить, где какой).
- **Location Transparency** — клиенту для работы не нужно знать физическое местоположение сервера (этим должно заниматься middleware).

Клиенты должны принимать во внимание наличие задержек:

- Нужно понимать разницу между отказом сети и отказом сервиса (и использовать Exponential backoff при повторных запросах, иначе рискуем вместе с сервисом положить и сервер).
- Клиенты должны знать о задержках при передаче данных (чтобы у них при необходимости была возможность прервать вызов и работать по альтернативному сценарию).



**RMI** (*remote method invocation*) - продолжение идей RPC. Например, Java-RMI.

Получаем доступ к удаленному объекту и вызываем метод.

TLDR - RPC - структурное программирование, RMI - ООП.

**Protocol buffers** (protobuf) - протокол, предложенный Google. Используется для сериализации данных. Более эффективен чем XML.

**gRPC** - RPC система от Google, разработанная в 2015 году. Использует protobuf для описания интерфейса взаимодействия.

В gRPC есть 4 вида вызовов:

- Один запрос — одно сообщение в ответ (обычный вызов функции)
- Один запрос — поток сообщений в ответ (клиенту возвращается поток, из которого он читает последовательность сообщений)
- Поток запросов — одно сообщение в ответ
- Двухнаправленный поток запросов (у сервера и клиента есть read/write stream)

Технические подробности:

- Сервисы описываются в том же .proto-файле, что и протокол protobuf-a
- В качестве типов параметров и результатов — message-и protobuf-a

Интерфейс описывается с помощью protocol buffer, затем специальный компилятор генерирует по нему код для клиента и сервера (на стороне сервера активно ожидаются запросы, а на стороне клиента вызываются методы у заглушки).

В большинстве языков для gRPC поддерживается и синхронная, и асинхронная обработка запросов.

## 56. Веб-сервисы, SOAP. WCF.

**Веб-сервис** — программная система с интерфейсами. HTTP-запросы (асинхронно, запрос-ответ, событийная схема), XML/JSON формат сообщений. Пример - авиакомпания, которые предоставляют информацию о билетах бюро путешествий.

**Web-services** (определение The Stencil Group) — свободно связанные, многократно используемые и, кроме того, распределенные компоненты, которые семантически инкапсулируют дискретную функциональность, и к которым можно программно обращаться посредством стандартных Интернет-протоколов.

При этом сервис обладает свойствами:

- Обнаруживаемость – должен обладать идентификатором (URI), по которому можно идентифицировать сервис и получить к нему доступ.
- Самоописание – сервис может предоставить документ с его контрактом.

HTTP-запрос для выполнения команды:

- Асинхронное взаимодействие
- Ответ-запрос
- Событийные схемы

XML или JSON как основной формат сообщений:

- SOAP/WSDL/UDDI
- XML-RPC
- REST

SOAP-ориентированные сервисы:

**SOAP** — Simple Object Access Protocol - обмен сообщениями в формате XML.

**WSDL** — Web Services Description Language - язык описания интерфейсов (на основе XML).

**UDDI** — Universal Description Discovery and Integration.

Описывает способ опубликования и обнаружения информации о Web-службах. Покров UDDI-регистра (UDDI registry cloud), или бизнес-регистр, предоставляет доступ к информации о Web-службах по схеме "зарегистрируйся один раз, будешь опубликован везде".



Преимущества:

- + В силу серьезной стандартизации поддерживается высокая степень автоматизации (описания сервисов, поддержки описаний, валидации сообщений)
- + Работает через HTTP (можно хоть GET использовать)

**WCF** (Windows Communication Foundation) — фреймворк для обмена данными:

- Входит в .NET Framework
- Умеет в WSDL, SOAP и тд.
- Автоматическая генерация заглушек на стороне клиента.

**ABCs of WCF** — акроним, включающий в себя необходимые для описания WCF endpoint понятия:

- **A — Address.** Где можно найти сервис (IP-адрес, URL, имя сервера и т.д.).
- **B — Binding.** Как endpoint взаимодействует с другими endpoint'ами. Как минимум, необходимо указать используемый транспортный протокол (TCP или HTTP).
- **C — Contract.** Соглашение между клиентом и сервером о структуре передаваемых сообщений.

## 57. Очереди сообщений, RabbitMQ.

**Очередь сообщений** — гарантирует доставку сообщений. Есть локальное хранилище у клиентов. Обычно — через “издатель - подписчик” (но может работать и в режиме “точка-точка”).

**Брокер сообщений** — преобразует сообщения по одному протоколу от клиента в сообщение по протоколу приемника, а также проверяет их на ошибки, фильтрует, маршрутизирует, вызывает веб-сервисы и хранит.

### **RabbitMQ:**

- Сервер и клиенты системы надежной передачи сообщений
- Сообщение посылается на сервер и хранится там, пока его не заберут
- Продвинутое возможности по маршрутизации сообщений
- Реализует протокол **AMQP** (Advanced Message Queuing Protocol), но может использовать и другие протоколы
- Сервер написан на Erlang, клиентские библиотеки доступны для практически чего угодно

**AMQP** — открытый протокол для передачи сообщений между компонентами системы. Основная идея состоит в том, что отдельные подсистемы (или независимые приложения) могут обмениваться произвольным образом сообщениями через AMQP-брокер, который осуществляет маршрутизацию, возможно гарантирует доставку, распределение потоков данных, подписку на нужные типы сообщений.

## 58. REST.

**REST** — Архитектурный стиль взаимодействия компонентов распределенного приложения в сети. Representational State Transfer.

Требования к REST-системам:

- Клиент-сервер
- Отсутствие состояния (запросы составляются так, чтобы сервер получал всю необходимую информацию для их выполнения)
- Кэширование (возможно)
- Единообразие интерфейса
  - Идентификация ресурсов (например, через URI)
  - Манипуляция ресурсами через представление (для описания запроса достаточно HTTP-метода и URI)
  - «Самоописываемые» сообщения (по самому сообщению понятно, как его обрабатывать)
  - Гипермедиа как средство изменения состояния приложения (клиент знает только о простых конечных точках; об остальных узнает из запросов)
- Слои (промежуточные узлы, клиент не может знать, связан ли он с сервером напрямую или нет)

Ресурс может быть одним **элементом** или **коллекцией**.

Например, *customers* — это ресурс-коллекция, а *customer* — ресурс-элемент. Мы можем идентифицировать ресурс коллекции *customers*, используя URI `/customers`, а ресурс одного клиента *customer* с помощью URI `/customers/{customerId}`.

Общение с помощью HTTP-методов:

- GET
- PUT
- POST
- DELETE

Достоинства:

- Простота
- Надежность
- Производительность
- Масштабируемость
- Легкость внесения изменений



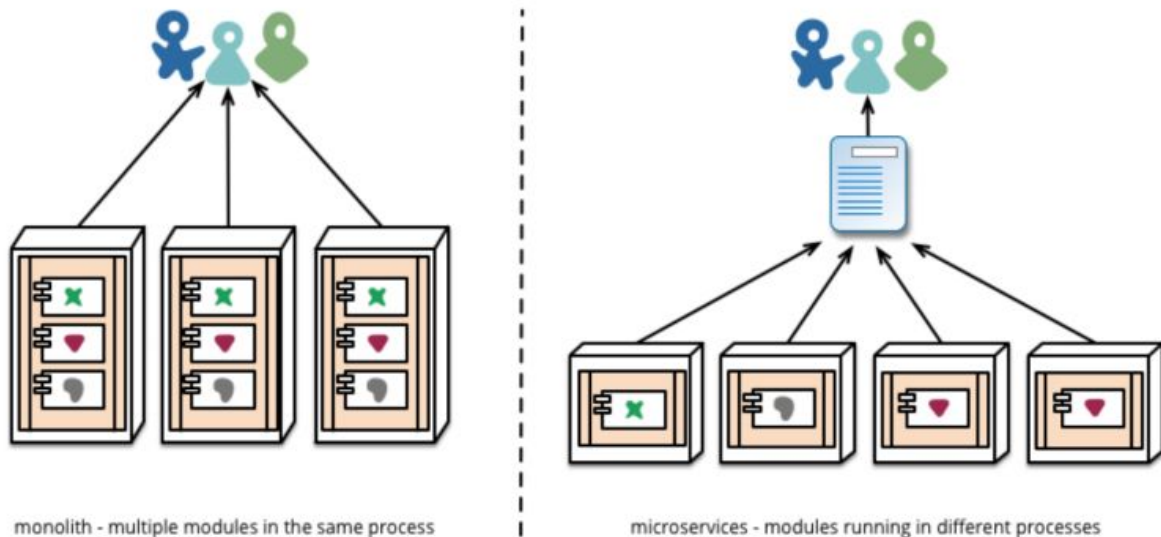
## 59. Микросервисы, peer-to-peer.

Микросервисы - набор небольших сервисов с использованием разных языков и технологий.

- Каждый в собственном процессе
  - Независимое развертывание
  - Децентрализованное управление
- Легковесные коммуникации

Недостатки монолитных приложений:

- Большой и сложный MVC (Model-View-Controller)
- Единый процесс разработки и стек технологий
- Сложная архитектура
- Сложно масштабировать
- Сложно вносить изменения



Недостатки микросервисов:

- Сложности выделения границ сервисов
- Перенос логики на связи между сервисами
  - Большой обмен данными
  - Нетривиальные зависимости
- Нетривиальная инфраструктура
- Нетривиальная переиспользуемость кода

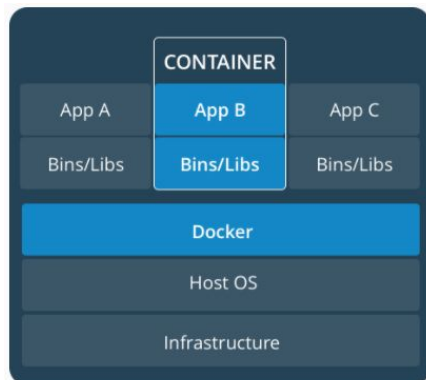
P2P:

- Децентрализованный и самоорганизующийся сервис
- Динамическая балансировка нагрузки
  - Вычислительные ресурсы
  - Хранилища данных
- Динамическое изменение состава участников

Пример: Bittorrent. Трекеры, метаданные, обмен сегментами. Есть бестрекерная реализация (это про DHT, полагаю).

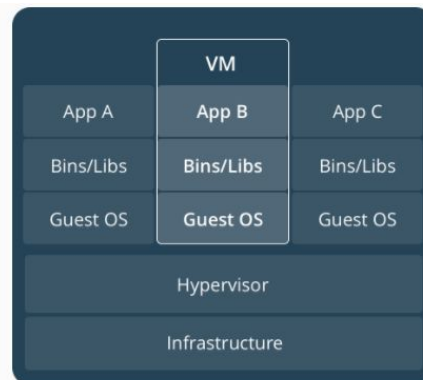
## 60. Развертывание и балансировка нагрузки, Docker.

**Docker** - легковесная (хех) виртуальная машина. Очень популярен, есть публичный репозиторий, DSL для образов.



CONTAINERS

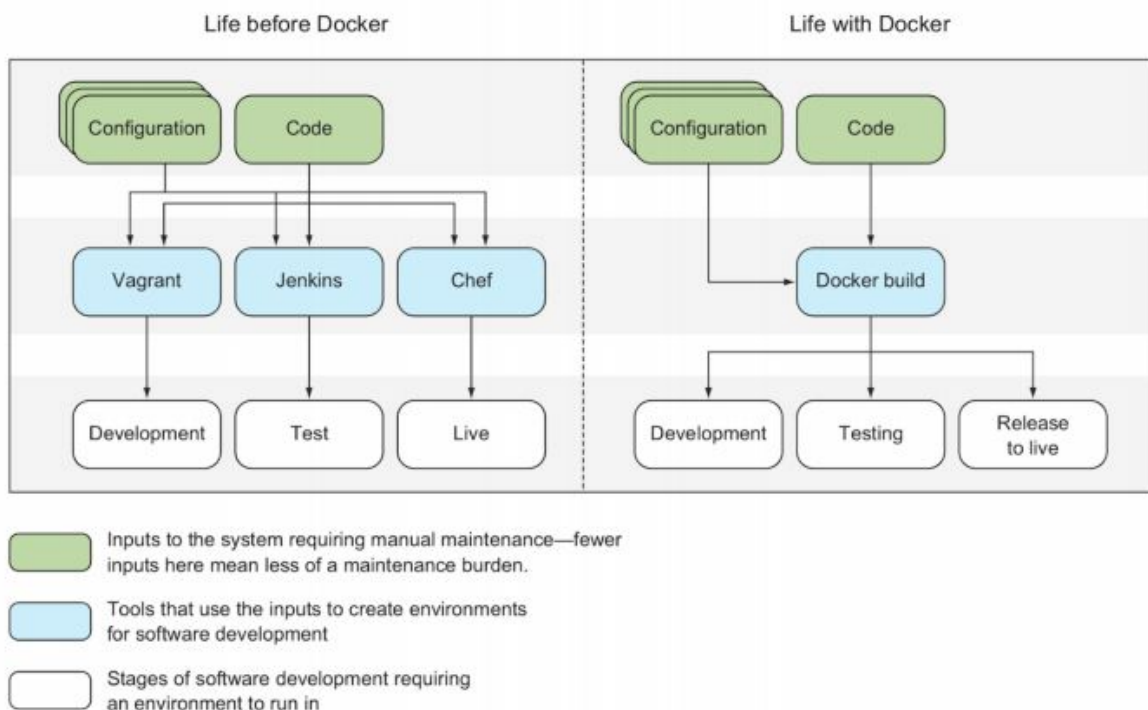
Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly.



VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

Образы легковеснее, быстрее и удобнее, но у них [меньше](#) изоляции, чем у VM.



**Docker image** (образ) - состоит из слоев. Все слои Read-only. Слои могут делиться между образами. На основе одного образа создается другой. У контейнера есть write слой. Содержит в себе один запущенный процесс.

Docker - клиент-серверное приложение. Есть Docker daemon - сервер по REST API, который управляет запущенными процессами и репозиторием образов. Docker Client - CLI для докера.

**DockerHub** - внешний репозиторий образов. Содержит официальные образы, пользовательские и приватные репозитории.

- `docker run` — запускает контейнер (при необходимости делает pull)
  - `-d` — запустить в фоновом режиме
  - `-p host_port:container_port` — прокинуть порт из контейнера на хост
  - `-i -t` — запустить в интерактивном режиме
  - Пример: `docker run -it ubuntu /bin/bash`
- `docker ps` — показывает запущенные контейнеры
  - Пример: `docker run -d nginx`; `docker ps`
- `docker stop` — останавливает контейнер (шлёт SIGTERM, затем SIGKILL)
- `docker exec` — запускает дополнительный процесс в контейнере

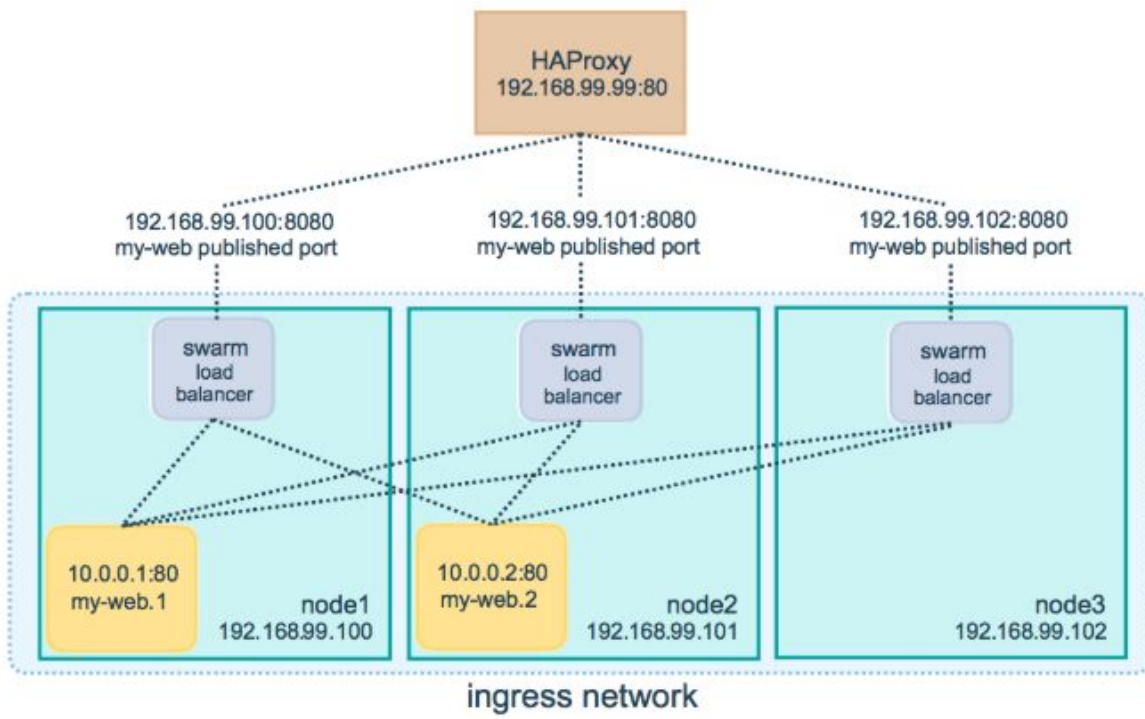
Команды `dockerfile`:

- `FROM` — взять как начальный образ указанный
- `WORKDIR` — выбрать папку в качестве текущей
- `RUN` — выполнить команду
- `ADD` — копировать файл/папку/архив
- `ENV` — установить `environment` переменную
- `ENTRYPOINT` — установить команду, которая выполнится при запуске контейнера
- `CMD` — установить принятые по умолчанию аргументы
- `EXPOSE` — разрешить доступ к контейнеру по порту
- `VOLUME` — определить mount point для volume
- `ARG` — build аргументы

Docker Compose используется для одновременного управления несколькими контейнерами, входящими в состав приложения. Управляется через `docker-compose.yml`. Подробнее [тут](#).

Docker swarm — распределенные контейнеры. Машина, на которой запускается контейнер, становится главной. Другие машины могут присоединяться к swarm-у и получать копию контейнера. Docker балансирует нагрузку по машинам. Любая нода способна обработать запрос к любому из сервисов кластера. [Подробнее](#).

[Про разницу](#).



## Вопросы

1. 30 билет “Моделирование ограничений”. В презентациях только про спецификацию, этого достаточно?
2. 24 билет, *Последовательное исполнение*